Available online at www.sciencedirect.com
ScienceDirect
Science of Computer Programming 64 (2007) 286–311
Science of Computer Programming
www.elsevier.com/locate/scico

Procedures and parameters in the real-time program refinement calculus

Ian J. Hayes

School of Information Technology and Electrical Engineering, The University of Queensland, Brisbane, 4072, Australia

Received 4 March 2005; received in revised form 13 June 2006; accepted 14 June 2006
Available online 8 August 2006

Abstract

The real-time refinement calculus is a formal method for the systematic derivation of real-time programs from real-time specifications in a style similar to the non-real-time refinement calculi of Back and Morgan. In this paper we extend the real-time refinement calculus with procedures and provide refinement rules for refining real-time specifications to procedure calls. A real-time specification can include constraints on, not only what outputs are produced, but also when they are produced. The derived programs can also include time constraints on when certain points in the program must be reached; these are expressed in the form of deadline commands. Such programs are machine independent. An important consequence of the approach taken is that, not only are the specifications machine independent, but the whole refinement process is machine independent. To implement the machine independent code on a target machine one has a separate task of showing that the compiled machine code will reach all its deadlines before they expire.

For real-time programs, externally observable input and output variables are essential. These differ from local variables in that their values are observable over the duration of the execution of the program. Hence procedures require input and output parameter mechanisms that are references to the actual parameters so that changes to external inputs are observable within the procedure and changes to output parameters are externally observable. In addition, we allow value and result parameters. These may be auxiliary parameters, which are used for reasoning about the correctness of real-time programs as well as in the expression of timing deadlines, but do not lead to any code being generated for them by a compiler.

Keywords: Procedures; Parameters; Real-time programming; Refinement calculus; Machine independent; Platform independent; Auxiliary variables; Auxiliary parameters; Deadlines

1. Introduction

Procedures are an important program structuring mechanism. They allow one to factor out a coherent piece of code and treat it as an abstract operation. Parameterisation allows further abstraction, as well as reuse of the procedure in a greater number of contexts. In the context of the refinement calculus [1,29] the specification of a procedure can be given in the form of an assumption plus a specification command. This allows one to separate the concerns of deriving a program that uses a procedure, where only its specification is needed, from the concerns of implementing the procedure.

E-mail address: Ian.Hayes@itee.uq.edu.au.

0167-6423/$ - see front matter
doi:10.1016/j.scico.2006.06.002

In this paper we consider adding procedures to the real-time refinement calculus [12,15,16,11,10]. The real-time context introduces a number of issues to do with reasoning about time. We begin with a discussion of the main issues via a simple example. The procedure *sample* (below) samples an integer input parameter, i, within a time interval between the time the procedure starts and an upper time limit, d. It returns the sample via the result parameter r. The input i is modelled as a trace over time, so that $i(t)$ represents the value of the input at time t. The procedure has an assumption (within the braces) that the input does not change (i.e., it is constant) over the closed time interval from the current time to time d. The current time is denoted by the special variable τ, in a manner similar to that used by Hehner [18]. The closed interval is denoted $[\tau \mathinner{..} d]$.

procedure *sample*(**input** $i : int$, **value aux** $d : Time$, **result** $r : int$)
$\{const(i, [\tau \mathinner{..} d])\}; r{:}[\exists t : [\tau_0 \mathinner{..} d] \bullet r = i(t)]$

The specification of the procedure body consists of the assumption that i is constant followed by a specification command that has a frame of r (that is, the only variable it can modify is r) and a postcondition (within the square brackets) that the final value of r is equal to the value of i at some time in the interval $[\tau_0 \mathinner{..} d]$, where τ_0 represents the time at which the specification command starts execution (which also corresponds to the value of τ used in the assumption). Because i is constant over the interval, it does not matter at which time i is sampled. The predicate *const* is defined as follows,[1]

$$const(E, S) \mathrel{\widehat{=}} S \neq \{\} \Rightarrow (\exists x \bullet (\forall t : S \bullet E \mathbin{@} t = x)),$$

where S is a set of times and E is an expression. The term $E \mathbin{@} t$ stands for the value of the expression E at time t. It is equivalent to the expression E with every occurrence of an input or output variable v replaced by $v(t)$ and every occurrence of τ replaced by t. Hence $const(i, [\tau \mathinner{..} d])$ is equivalent to

$$\begin{aligned} &[\tau \mathinner{..} d] \neq \{\} \Rightarrow (\exists x \bullet (\forall t : [\tau \mathinner{..} d] \bullet i(t) = x)) \\ \equiv\ &\tau \leq d \Rightarrow (\exists x \bullet (\forall t : Time \bullet \tau \leq t \leq d \Rightarrow i(t) = x)) \end{aligned}$$

The body of procedure *sample* can be implemented by the code,

call $read(i, r)$;
deadline d

which reads the value of i into r. The read is followed by a **deadline** command to ensure that the value is read before time d. The procedure *read* has the following specification.

procedure *read*(**input** $i : int$, **result** $r : int$)
$r{:}[\exists t : [\tau_0 \mathinner{..} \tau] \bullet r = i(t)]$

That is, it assigns to r the value of i at some time during its execution.

The **deadline** command [5,15] takes no time to execute and always guarantees to terminate by time d, even if execution of the deadline command starts after time d! Obviously such a command cannot be implemented in the normal manner (of generating machine code). In order to compile such programs for a particular target machine, we need to ensure that all paths leading to a deadline will reach it before the deadline expires; we return to this issue below.

An important consequence of the introduction of the deadline command is that it allows one to write real-time programs in a machine-independent form, and hence, as specifications are machine independent, the whole refinement process from specification to code is machine independent. The only phase of the program development process that is machine dependent is checking that the compiled machine code for a particular target machine reaches all the deadlines before they expire.

Consider the following code which calls the procedure *sample*. It assumes that the current time, τ, is before some time T and that the input *in* is constant from the current time until time $T + 10$ milliseconds (abbreviated 10 ms).

$\{\tau \leq T \wedge const(in, [\tau \mathinner{..} T + 10\ \text{ms}])\}$;
call $sample(in, T + 10\ \text{ms}, res)$

[1] Earlier papers use the term "stable" rather than "constant" but a referee suggested the change of terminology to avoid confusion with the use of the term "stable" by other authors.

One issue that arises in the real-time case (but not the standard case) is that, although the condition $\tau \leq T$ is assumed to hold immediately before the call on *sample*, it cannot be assumed to still hold at the time the body of *sample* begins execution. This is because the time taken to enter the procedure may take the current time to some time τ' greater than T. Fortunately, the same does not apply to the other assumption because, if $const(in, [\tau \mathinner{..} T + 10 \text{ ms}])$ holds, then $const(in, [\tau' \mathinner{..} T + 10 \text{ ms}])$ holds for any time τ' greater than or equal to τ, because $[\tau' \mathinner{..} T + 10 \text{ ms}]$ is a subset of $[\tau \mathinner{..} T + 10 \text{ ms}]$. Predicates that have this property are referred to as being *idle invariant* [16] and play a significant role in the refinement rules presented in this paper.

Because the time immediately before the call on *sample* is assumed to be less than or equal to T, in order for the deadline within the implementation to be reached before it expires at time $T + 10$ ms, the execution path consisting of the procedure call entry followed by the read must be executed in less than 10 ms on the target machine. This can be checked by performing a worst-case execution-time analysis [4,27] on the code generated for this path for the target machine.

Before we leave our sample program, it is worth commenting on the role of the auxiliary parameter d. Within the implementation, d is only used in the deadline command, for which no code is generated. Hence when generating code for the procedure the parameter d does not have to be explicitly passed. It is only used for the timing analysis discussed above. Because such auxiliary timing parameters are useful for specifying real-time programs, we have included them in our language. Auxiliary parameters can only be used in assumptions, specification commands, deadline commands, and in assignments to other auxiliary variables. These constraints ensure that there is never any need to generate code for constructs involving auxiliary variables [9].

Hooman and Van Roosmalen have developed a platform-independent approach similar to that used in this paper [23]. Their approach makes use of timing annotations that are associated with commands. The annotations allow the capture in auxiliary timing variables of the time of occurrence of significant events that occur with the associated command, and the expression of timing deadlines on the command relative to such timing variables.

The approach using deadlines is more general than the approach of Shaw [32] which requires a timing constraint on every command. Using deadlines, constraints only need to be placed where necessary, and they constrain the whole path leading to the deadline. Having timing constraints on every command significantly complicates the refinement process and may overconstrain the implementation.

Recursion. One issue with recursion is that there is no *a priori* bound on the depth of nesting of recursive calls, and hence no limit on the stack space required. For real-time applications, especially safety-critical ones, the possibility of stack overflow is problematic. For this reason many real-time programming approaches ban recursion [2,3]. This problem may be avoided if a constant bound on the depth of recursion can be determined for the program.

Recursion also introduces an additional problem for timing path analysis if a path crosses a multiple number of procedure entry and/or exit boundaries. For example, when a tail recursive procedure (in which the last action of the procedure is a recursive call on itself) exits its deepest level of call, it will return through all intermediate calls. The fact that the number of levels of calls is variable, implies that the time constraint on the path must also allow for the variable number of procedure exits. Such variable time constraints complicate timing analysis. Fortunately in this case, tail recursion may be eliminated and replaced by a repetition.

In this paper we follow the lead of Spark Ada [2] and the Ravenscar Tasking Profile [3] and ban recursion. This also sidesteps the issue of showing termination of recursive procedures. Recursion for non-real-time procedures is treated by Hesselink [20] and Staples [34].

Section 2 introduces a machine-independent, wide-spectrum language used for specification, and refinement to code. Section 3 defines procedures and Section 4 gives a set of refinement laws for procedures. Section 5 discusses timing constraint analysis.

2. Wide-spectrum language

In this section we introduce our wide-spectrum language, its semantics, and give a number of refinement laws. The semantics are adapted from previous work [12] based on a predicative approach similar to that of Hehner [19]. Readers familiar with the approach can skip to Section 3 and use Figs. 1–4 for reference. We model time by non-negative real numbers, including infinity to allow for nonterminating programs:

$$Time_\infty \mathrel{\widehat{=}} \{r : \mathbb{R} \mid 0 \leq r\} \cup \{\infty\}$$
$$Time \mathrel{\widehat{=}} \{t : Time_\infty \mid t < \infty\}$$

We use continuous time because it allows one to specify operations like sampling an analog input. Using continuous time one can specify assumptions like the rate of change of an analog input is bounded (using the derivative of the input with respect to time). In addition, outputs of control systems are observable over all time, not just discrete points of time, and the continuous model better reflects this aspect.

We use the term *environment* to refer to the identifiers that are in scope and their definitions/types; an environment, ρ, contains the following kinds of identifiers [35]:

- inputs, $\rho.in$, which are under external control;
- outputs, $\rho.out$, which are under the control of the program;
- local variables, $\rho.var$, which are under the control of the program, but unlike outputs are not externally visible;
- local auxiliaries, $\rho.aux$, which are similar to local variables, but are restricted to appear only in assumptions, specifications, and expressions used in deadline commands, assignments to auxiliaries, and as actual auxiliary parameters in calls;
- procedures, $\rho.proc$; and
- the current time variable, τ.

An environment records the type of each variable and the definition of each procedure. Because typing is similar to the standard case we do not treat it in detail here. The definition of a procedure consists of its formal parameters and its body. The current time variable, τ, acts like an auxiliary, but because of its special nature we do not include it in $\rho.aux$. Inputs and outputs are modelled as functions from *Time* to the declared type of the variable, e.g., given the declaration,

input *sensor* : Boolean,

sensor is modelled as a function from *Time* to Boolean, with $sensor(t)$ giving the value of *sensor* at time t. Note that it is not meaningful to talk about the value of a variable at time infinity, even though we allow the current time variable, τ, to take on the value infinity to indicate nontermination.

Semantics. The semantic domains are summarised in Fig. 1. For an environment ρ, In_ρ stands for the type of the inputs in that environment. Outputs are modelled as partial functions from time to allow for the fact that their values at a particular time, τ, are only defined up until τ. We use $\rho.local$ to refer to all the local variables, including auxiliaries. The type of the local variable state is $Local_\rho$. The overall state, Σ_ρ, of a program execution is defined as a Z schema[2] [33,8] consisting of the local state (*loc*), the current time (τ), a Boolean component representing that the program has not aborted (*ok*), and the traces of the outputs up until time τ (*out*). *Expressions* and *single-state predicates* ($Pred_\rho$) are defined over inputs and the (overall) state, and "$\Rrightarrow$" is defined as implication over all input and state values. *Relational predicates* ($Rel_{\rho,\rho'}$), or *relations* for short, are defined over inputs (denoted by *in*), and pre-states and post-states (denoted by σ_0 and σ, respectively). Implication ($\Rrightarrow$) over all inputs, pre-states and post-states is also defined for relations. We allow a single-state predicate, P, to be used in a context in which a relation is expected, in which case it constrains the post-state, σ, and ignores the pre-state, σ_0. That is, P is treated as the following relation: $(\lambda\ in : In_\rho; \sigma_0 : \Sigma_\rho; \sigma : \Sigma_\rho \bullet P(in, \sigma))$, where this relation has ρ as both its before and after environments.

In actual specifications and laws a relation over inputs *in*, before state σ_0 and after state σ is written using the conventional notation [29] in which $\sigma.loc(y)$ is represented by y, $\sigma_0.loc(y)$ by y_0, $\sigma.\tau$ by τ, $\sigma_0.\tau$ by τ_0, $\sigma.out(o)$ by o, and $in(i)$ by i. Single-state predicates are expressed in a similar manner, but do not use initial (zero-subscripted) variables.

Because an execution path may start in an environment ρ and cross the start of a local variable block but not its end (or vice versa), the end of the path may have a different environment ρ'. We use a function, *EnvOut*, that for any command C and environment ρ, in which C is well defined, returns the final environment, $EnvOut(C)(\rho)$. In the definition of commands below we assume the before and after environments are the same, unless otherwise stated. The set of well-defined commands in an environment ρ is given by $VCommand_\rho$.

The meaning of a command, C, is given a function, $\mathcal{M}$, that for an environment ρ, that is in the domain of $EnvOut(C)$, returns a relation, $\mathcal{M}_\rho(C) \in CommandRel_{\rho,\rho'}$ between inputs, source state Σ_ρ and target state $\Sigma_{\rho'}$,

[2] A schema is similar to a record, but can include a constraint on its components.

Val stands for the universal set of values that variables may take on. Given an environment, ρ, let $P_1, P_2 \in Pred_\rho$; $R_1, R_2 \in Rel_{\rho,\rho'}$; and $C_1, C_2 \in VCommand_\rho$. For a function or relation f, $\mathrm{dom}(f)$ stands for its domain and $\mathrm{ran}(f)$ stands for its range. The infix operator "\" is set difference.

$$In_\rho \mathrel{\widehat{=}} \rho.in \to (Time \to Val)$$
$$Out_\rho \mathrel{\widehat{=}} \rho.out \to (Time \nrightarrow Val)$$
$$\rho.local \mathrel{\widehat{=}} \rho.var \cup \rho.aux$$
$$Local_\rho \mathrel{\widehat{=}} \rho.local \to Val$$
$$\Sigma_\rho \mathrel{\widehat{=}} [loc : Local_\rho; \tau : Time_\infty; ok : \mathrm{Boolean}; out : Out_\rho | \forall o : \mathrm{ran}(out) \bullet \mathrm{dom}(o) = [0 \mathinner{..} \tau] \setminus \{\infty\}]$$
$$Expr_\rho \mathrel{\widehat{=}} In_\rho \times \Sigma_\rho \to Val$$
$$Pred_\rho \mathrel{\widehat{=}} In_\rho \times \Sigma_\rho \to \mathrm{Boolean}$$
$$P_1 \Rrightarrow P_2 \mathrel{\widehat{=}} (\forall in : In_\rho; \sigma : \Sigma_\rho \bullet P_1(in, \sigma) \Rightarrow P_2(in, \sigma))$$
$$Rel_{\rho,\rho'} \mathrel{\widehat{=}} In_\rho \times \Sigma_\rho \times \Sigma_{\rho'} \to \mathrm{Boolean}$$
$$R_1 \Rrightarrow R_2 \mathrel{\widehat{=}} (\forall in : In_\rho; \sigma_0 : \Sigma_\rho, \sigma : \Sigma_{\rho'} \bullet R_1(in, \sigma_0, \sigma) \Rightarrow R_2(in, \sigma_0, \sigma))$$
$$EnvOut : Command \to (Env \nrightarrow Env)$$
$$VCommand_\rho \mathrel{\widehat{=}} \{C : Command | \rho \in \mathrm{dom}(EnvOut(C))\}$$
$$out1 \text{ prefix } out2 \mathrel{\widehat{=}} (\forall o : \mathrm{dom}(out1) \cap \mathrm{dom}(out2) \bullet out1(o) \subseteq out2(o))$$
$$CommandRel_{\rho,\rho'} \mathrel{\widehat{=}} \{R : Rel_{\rho,\rho'} | \forall in : In_\rho; \sigma_0 : \Sigma_\rho; \sigma : \Sigma_{\rho'} \bullet$$
$$R(in, \sigma_0, \sigma) \Rightarrow \sigma_0.\tau \le \sigma.\tau \wedge (\sigma_0.ok \wedge \sigma_0.\tau = \infty \Rightarrow \sigma.ok) \wedge \sigma_0.out \text{ prefix } \sigma.out\}$$
$$C_1 \sqsubseteq_\rho C_2 \mathrel{\widehat{=}} (EnvOut(C_1)(\rho) = EnvOut(C_2)(\rho)) \wedge (\mathcal{M}_\rho(C_2) \Rrightarrow \mathcal{M}_\rho(C_1))$$
$$C_1 \sqsupseteq\sqsubseteq_\rho C_2 \mathrel{\widehat{=}} (C_1 \sqsubseteq_\rho C_2) \wedge (C_2 \sqsubseteq_\rho C_1)$$

Fig. 1. Semantic domains.

where $\rho' = EnvOut(C)(\rho)$. We place three constraints on the relation: time cannot go backwards; if the program preceding the command neither aborted (i.e., $\sigma_0.ok$) nor terminated (i.e., $\sigma_0.\tau = \infty$) then the command does not abort; and the initial value of each output trace must be a prefix of its final value (the prefix is denoted by "$\subseteq$"). Given commands C_1 and C_2 that are well defined in an environment ρ, C_1 is refined by C_2 (denoted $C_1 \sqsubseteq_\rho C_2$) if they have the same final environment and the relation defined by C_2 is contained in that defined by C_1. Refinement equivalence is denoted by $C_1 \sqsupseteq\sqsubseteq_\rho C_2$.

Fundamental commands. We define a possibly nonterminating real-time *specification command*, $\infty\vec{x}{:}[Q]$, similar to that of Morgan [29], in which $\vec{x}$ is a vector of variables called the *frame*, and the relation Q is its postcondition. Q may only reference variables that are in the environment, ρ, as well as constants and initial variable counterparts of local variables. The special variable *ok* cannot be referenced by Q. Because τ may take on the value infinity, the specification command allows nontermination. The '∞' at the beginning is just part of the syntax; there is also a terminating specification command (see Fig. 3) that does not have the '∞' at the front. If the specification command does not terminate, there is no final state for the local variables. Hence we require that the postcondition Q does not constrain the final values of the local variables in this case: it is *nontermination state independent*.

Definition 1 (*Nontermination State Independent*). In an environment ρ a relation Q is *nontermination state independent* if,

$$\tau = \infty \Rrightarrow (Q \Leftrightarrow (\forall \rho.local \bullet Q)).$$

The final environment of a specification command is the same as its initial environment. The semantics of a specification command is given in Fig. 2 in terms of a function $\mathcal{M}_\rho(\infty\vec{x}{:}[Q])$ that for an environment ρ gives the semantics of the command as a relation of type $CommandRel_{\rho,\rho}$. The frame, $\vec{x}$, of a specification command lists

Let ρ be an environment; $P \in Pred_\rho$; $Q \in Rel_{\rho,\rho}$; $\vec{x}$ consist of outputs and locals in ρ; $C_1 \in VCommand_\rho$; and $C_2 \in VCommand_{\rho'}$, where $\rho' \mathrel{\widehat{=}} EnvOut(C_1)(\rho)$. We require that P and Q are independent of ok. Let $\vec{no}$ stand for the vector of outputs in $\rho.out$ but excluding outputs in $\vec{x}$, and $\vec{nx}$ stand for the vector of local variables $\rho.var$ and auxiliaries $\rho.aux$ but excluding variables in $\vec{x}$.

$$
\begin{array}{l}
EnvOut(\infty \vec{x}: [Q])(\rho) \mathrel{\widehat{=}} \rho \\
\mathcal{M}_\rho\ (\infty \vec{x}: [Q]) \mathrel{\widehat{=}} (\lambda\ in : In_\rho; \sigma_0, \sigma : \Sigma_\rho \bullet \sigma_0.\tau \le \sigma.\tau \wedge \\
\quad (\sigma_0.ok \wedge \sigma_0.\tau = \infty \Rightarrow \sigma.ok) \wedge \sigma_0.out \text{ prefix } \sigma.out \wedge \\
\quad (\sigma_0.ok \wedge \sigma_0.\tau < \infty \Rightarrow \sigma.ok \wedge Q(in, \sigma_0, \sigma) \wedge const(\vec{no}, [\sigma_0.\tau \mathinner{..} \sigma.\tau]) \wedge (\sigma.\tau < \infty \Rightarrow \vec{nx}_0 = \vec{nx}))) \\
EnvOut(\{P\})(\rho) \mathrel{\widehat{=}} \rho \\
\mathcal{M}_\rho\ (\{P\}) \mathrel{\widehat{=}} (\lambda\ in : In_\rho; \sigma_0, \sigma : \Sigma_\rho \bullet \sigma_0.\tau \le \sigma.\tau \wedge \\
\quad (\sigma_0.ok \wedge \sigma_0.\tau = \infty \Rightarrow \sigma.ok) \wedge \sigma_0.out \text{ prefix } \sigma.out \wedge \\
\quad (\sigma_0.ok \wedge \sigma_0.\tau < \infty \wedge P(in, \sigma_0) \Rightarrow \sigma_0 = \sigma)) \\
EnvOut(C_1; C_2)(\rho) \mathrel{\widehat{=}} EnvOut(C_2)(EnvOut(C_1)(\rho)) \\
\mathcal{M}_\rho\ (C_1; C_2) \mathrel{\widehat{=}} \textbf{let}\ \rho' \mathrel{\widehat{=}} EnvOut(C_1)(\rho) \bullet \textbf{let}\ \rho'' \mathrel{\widehat{=}} EnvOut(C_2)(\rho') \bullet \\
\quad (\lambda\ in : In_\rho; \sigma_0 : \Sigma_\rho; \sigma : \Sigma_{\rho''} \bullet (\exists \sigma' : \Sigma_{\rho'} \bullet \mathcal{M}_\rho\ (C_1)\ (in, \sigma_0, \sigma') \wedge \mathcal{M}_{\rho'}\ (C_2)\ (in, \sigma', \sigma)))
\end{array}
$$

Fig. 2. Semantics of commands.

those variables in the environment ρ that may be modified by the command. An empty frame is indicated by $\varnothing$. The frame must not include inputs. The current time variable, τ, is implicitly in the frame. All outputs not in the frame, i.e., those in $\rho.out$ but not $\vec{x}$, are defined to be constant for the duration of the command. We allow the first argument of *const* to be a vector of variables, in which case all variables in the vector are constant.

Any local variable, y, not in the frame of a specification command is unchanged. Hence for these variables we require that $y_0 = y$, except that in the case of a nonterminating command there is no final state and hence the equality is not meaningful if the final time is infinity.

For a single-state predicate, $P \in Pred_\rho$, an *assumption*, $\{P\}$, states that the predicate P may be assumed to hold at that point in the program. Its semantics is given in Fig. 2. P must be independent of the special variable ok.

If a command C_1 in environment ρ has a corresponding final environment of ρ', and a command C_2 in environment ρ' has a corresponding final environment of ρ'', then the semantics of the sequential composition "$C_1; C_2$" is given by a relation of type $CommandRel_{\rho,\rho''}$ as defined in Fig. 2. The relation composes the semantics of the two commands by introducing an intermediate state σ'.

Primitive real-time commands. In Fig. 3 we define: a terminating specification command, $\vec{x}: [Q]$; the null command, **skip**, that does nothing and takes no time; a command, **idle**, that does nothing but may take any finite time; multiple assignments (assignments to auxiliaries take no time); and the deadline command.

Expressions that are evaluated at runtime, for example, those used in assignments to nonauxiliary variables or as nonauxiliary actual parameters within a procedure call or in guards of selections (**if**) or iterations (**do**) are considered to be a subset of those used in specifications. When expressions are used in specifications we assume they are total (i.e., they are defined for all values of their arguments[3]) but when expressions are evaluated at runtime, they may not be well defined, for example, if they contain a division by zero. For the subset of expressions evaluated at runtime the predicate $def(\vec{E})$ characterises those states in which the expressions $\vec{E}$ are well defined. Such expressions may not refer to zero-subscripted variables, but they may refer to the value of an output variable o at the current time via just the identifier o. Because outputs are constant while such expressions are being evaluated it does not matter at what time the output is examined. We often require that expressions are idle-constant, that is, their value does not change over time provided all the variables under the control of the program are constant.

[3] For example, a division by zero may give the result infinity, etc.

Definition 2 (*Primitive Real-time Commands*). Given a vector of variables, $\vec{x}$, not including any inputs; a relation, Q; a vector of variables, $\vec{y}$, not including any inputs; a vector of idle-constant expressions, $\vec{E}$, of the same length as $\vec{y}$ and whose type is assignment compatible with $\vec{y}$; a vector of auxiliary variables, $\vec{a}$; a vector of expressions, $\vec{F}$, of the same length as $\vec{a}$ and whose type is assignment compatible with $\vec{a}$; and a time-valued expression D; the primitive real-time commands are defined as below, where $\vec{E}_0$ and $\vec{F}_0$ stand for $\vec{E}$ and $\vec{F}$ with all occurrences of local and auxiliary variables replaced by their zero-subscripted counterparts. Recall that $E \mathbin{@} \tau_0$ stands for E with every occurrence of an input or output variable, v, replaced by $v(\tau_0)$, and every occurrence of τ replaced by τ_0.

$$\vec{x}: [Q] \mathrel{\widehat{=}} \infty\vec{x}: [Q \wedge \tau < \infty]$$
$$\mathbf{skip} \mathrel{\widehat{=}} \varnothing: [\tau_0 = \tau]$$
$$\mathbf{idle} \mathrel{\widehat{=}} \varnothing: [true]$$
$$\vec{y} := \vec{E} \mathrel{\widehat{=}} \{def(\vec{E} \mathbin{@} \tau)\}; \vec{y}: [\vec{y} \mathbin{@} \tau = (\vec{E}_0 \mathbin{@} \tau_0)]$$
$$\vec{a} := \vec{F} \mathrel{\widehat{=}} \vec{a}: [\vec{a} = (\vec{F}_0 \mathbin{@} \tau_0) \wedge \tau = \tau_0]$$
$$\mathbf{deadline}\ D \mathrel{\widehat{=}} \infty\varnothing: [\tau_0 = \tau \leq (D \mathbin{@} \tau_0)]$$

Fig. 3. Definition of primitive real-time commands.

Definition 3 (*Idle-constant*). An expression E over an environment ρ is *idle-constant* provided,

$$\tau_0 \leq \tau < \infty \wedge const(\rho.out, [\tau_0 \mathinner{..} \tau]) \Rrightarrow E \mathbin{@} \tau_0 = E \mathbin{@} \tau.$$

Theorem 4 follows from Definition 3 and the fact that outputs are constant [16].

Theorem 4. *If τ does not occur free in an expression E, and E contains no references to inputs, then E is idle-constant.*

Refinement laws. We give the properties we need of commands in the form of refinement laws in Fig. 4 [11,12]. A specification command may be refined by strengthening its postcondition. For the strengthening one may also assume the following: time does not go backwards; the start time is not infinity; that any immediately preceding assumption holds for the initial state; that outputs not in the frame are constant for the duration of the command; and if the command terminates, the final values of the local variables that are not in the frame are the same as their initial values.

Because we allow nonterminating commands, we need to be careful with the law for sequential composition. If the first command of the sequential composition does not terminate, then we want the overall effect of the sequential composition on the values of the outputs over time to be the same as just the effect of the first command. In Law 10 (sequential) in Fig. 4 this is achieved by guarding the use of R_2 by $\tau' < \infty$.

Local variables and auxiliaries. A block may introduce a new local variable or auxiliary, whose name must not already appear in the environment. We use the primitive command "**alloc var** w" to allocate a local variable w and "**alloc aux** w" to allocate an auxiliary w. The **alloc** primitive takes no time to execute, but expands the state space with the new variable, and leaves the existing variables unchanged. If w does not occur in ρ and ρ' is the same as ρ except that $\rho'.var = \rho.var \cup \{w\}$, then $EnvOut(\mathbf{alloc\ var}\ w)(\rho) = \rho'$. Allocating an auxiliary is similar except $\rho'.aux = \rho.aux \cup \{w\}$.

To deallocate a local variable or auxiliary we use the primitive "**dealloc** w". It also takes no time to execute, but removes w from the environment leaving the other variables unchanged.

An auxiliary variable block with body the command C, can be defined using **alloc** and **dealloc**. The initial value of w is an element of its type T. T must be nonempty.

$$|[\ \mathbf{aux}\ w : T; C\ ]| \mathrel{\widehat{=}} (\mathbf{alloc\ aux}\ w; w: [w \in T \wedge \tau = \tau_0]; C; \mathbf{dealloc}\ w)$$

The allocation and deallocation of a local variable may take time. This is modelled by the fact that the specification command used to ensure w is an element of its type may take time and by the use of an **idle** command after the

Given single-state predicates P, P_1, and P_2; relations Q, R, R_1 and R_2; disjoint sets of locals $\vec{x}$ and $\vec{v}$; disjoint sets of outputs $\vec{o}$ and $\vec{u}$; and commands C_1 and D_1; all over an environment ρ, and commands C_2 and D_2 over an environment ρ', where $\rho' \mathrel{\widehat{=}} \mathit{EnvOut}(C_1)(\rho)$, the laws below hold. P_0 stands for P with all occurrences of local variables and τ replaced by their zero-subscripted counterparts. Also let $\vec{no}$ stand for the vector of outputs $\rho.\mathit{out}$ but excluding $\vec{o}$, and $\vec{nx}$ stand for the vector of variables in $\rho.\mathit{var} \cup \rho.\mathit{aux}$ but excluding $\vec{x}$.

Law 5 (*Strengthen Postcondition*). *Provided*

$$\left(\tau_0 \le \tau \wedge \tau_0 < \infty \wedge P_0 \wedge \mathit{const}(\vec{no}, [\tau_0 \mathinner{..} \tau]) \wedge (\tau < \infty \Rightarrow \vec{nx}_0 = \vec{nx}) \wedge R\right) \Rrightarrow Q$$

then $\{P\}; \infty\vec{x}, \vec{o}{:}[Q] \sqsubseteq_\rho \{P\}; \infty\vec{x}, \vec{o}{:}[R]$.

Law 6 (*Contract Frame*).

$$\infty\vec{v}, \vec{u}, \vec{x}, \vec{o}{:}[Q \wedge \mathit{const}(\vec{o}, [\tau_0 \mathinner{..} \tau]) \wedge (\tau < \infty \Rightarrow \vec{x} = \vec{x}_0)] \sqsupseteq_\rho \infty\vec{v}, \vec{u}{:}[Q].$$

Law 7 (*Weaken Assumption*). *If* $P_1 \Rrightarrow P_2$ *then* $\{P_1\} \sqsubseteq_\rho \{P_2\}$.

Law 8 (*Remove Assumption*). $(\{P\}; C) \sqsubseteq_\rho C$.

Law 9 (*Post Assumption*). *If* $Q \Rrightarrow P$ *then* $\infty\vec{x}, \vec{o}{:}[Q] \sqsubseteq_\rho \infty\vec{x}, \vec{o}{:}[Q]; \{P\}$.

Law 10 (*Sequential*). *Provided*

$$\left(\begin{array}{l}\tau_0 < \infty \wedge P_0 \wedge \mathit{const}(\vec{no}, [\tau_0 \mathinner{..} \tau]) \wedge (\tau < \infty \Rightarrow \vec{nx}_0 = \vec{nx}) \wedge \\ \left(\exists \tau', \vec{x'} \bullet \tau_0 \le \tau' \le \tau \wedge R_1\left[\begin{smallmatrix}\tau', \vec{x'} \\ \tau, \vec{x}\end{smallmatrix}\right] \wedge \left(\tau' < \infty \Rightarrow R_2\left[\begin{smallmatrix}\tau', \vec{x'} \\ \tau_0, \vec{x}_0\end{smallmatrix}\right]\right)\right)\end{array}\right) \Rrightarrow Q$$

then $\{P\}; \infty\vec{x}, \vec{o}{:}[Q] \sqsubseteq_\rho \{P\}; \infty\vec{x}, \vec{o}{:}[R_1]; \infty\vec{x}, \vec{o}{:}[R_2]$.

Law 11 (*Monotonicity*). *Let* $\rho' \mathrel{\widehat{=}} \mathit{EnvOut}(C_1)(\rho)$, *if* $C_1 \sqsubseteq_\rho C_2$ *and* $D_1 \sqsubseteq_{\rho'} D_2$ *then* $C_1; D_1 \sqsubseteq_\rho C_2; D_2$.

Fig. 4. Refinement laws.

deallocation.

$$|[\ \mathbf{var}\ w : T; C\ ]| \mathrel{\widehat{=}} (\mathbf{alloc\ var}\ w; w{:}[w \in T]; C; \mathbf{dealloc}\ w; \mathbf{idle})$$

We require that w is not in ρ because, if it is, and was for example an output not in the frame of C, it would be required to be constant for the duration of C. If we allowed the newly declared local to replace w, this constraint would be missing.

When a new local or auxiliary variable is introduced, it may be modified by any command within its scope. For example, if a new local variable, w, is introduced with a scope that includes a specification command, $x{:}[Q]$, the frame of the specification command is extended with w, i.e., it becomes $w, x{:}[Q]$. In general, we use the notation $w : C$ to stand for the command C with the frame of every command within C expanded to include w. If C contains a primitive command, it may be necessary to convert the primitive command to an equivalent specification command in order to express $w : C$. For example, the command **idle** is equivalent to the specification command $\varnothing{:}[\mathit{true}]$, and hence w : **idle** is equivalent to $w{:}[\mathit{true}]$. Note that if w does not need to be modified this can be refined back to **idle**. Local and auxiliary variable blocks satisfy the following laws.

Law 12 (*Local and Auxiliary Variables*). *Given an environment,* ρ, *commands,* C, C_1 *and* C_2 *that are well defined in* ρ, *a fresh identifier,* w, *not in* ρ, *such that* w *does not occur free in* C_1 *and* C_2, *and a nonempty type* T:

$$\mathbf{idle}; C; \mathbf{idle} \sqsubseteq_\rho |[\ \mathbf{var}\ w : T; w : C\ ]| \quad \text{(a)}$$

$$C \sqsubseteq_\rho |[\ \mathbf{aux}\ w : T; w : C\ ]| \quad \text{(b)}$$

$$\mathbf{idle}; C_1; |[\ \mathbf{var}\ w : T; C\ ]|; C_2; \mathbf{idle} \sqsupseteq_\rho |[\ \mathbf{var}\ w : T; \mathbf{idle}; C_1; \mathbf{idle}; C; \mathbf{idle}; C_2; \mathbf{idle}\ ]| \quad \text{(c)}$$

$$C_1; |[\ \mathbf{aux}\ w : T; C\ ]|; C_2 \sqsupseteq_\rho |[\ \mathbf{aux}\ w : T; C_1; C; C_2\ ]| \quad \text{(d)}$$

We abbreviate multiple declarations by merging them into a single block, e.g., |[**var** $v : T$; |[**aux** $x : T'$; C]|]| = |[**var** $v : T$; **aux** $x : T'$; C]|. Other constructs such as selections and iterations are covered elsewhere [16,12,11].

Laws for substitutions. Before introducing parametrised procedures, we review the basic laws for substitution on predicates/relations. For distinct variables v and w, and terms e and f, a substitution of e for v within P can be written either $P\,[v\backslash e]$ or $P\left[\frac{e}{v}\right]$ and the following laws hold.

$$P\,[v\backslash e] \equiv P \quad \text{provided } v \text{ does not occur free in } P \tag{1}$$

$$P\left[\frac{w}{v}\right]\left[\frac{e}{w}\right] \equiv P\left[\frac{e}{v}\right] \quad \text{provided } w \text{ does not occur free in } P \tag{2}$$

$$P\left[\frac{e}{v}\right]\left[\frac{f}{w}\right] \equiv P\left[\frac{f}{w}\right]\left[\frac{e\left[\frac{f}{w}\right]}{v}\right] \quad \text{provided } v \text{ does not occur free in } f \tag{3}$$

$$\text{If } P \Rrightarrow Q \text{ then } P\,[v\backslash e] \Rrightarrow Q\,[v\backslash e]. \tag{4}$$

All the laws except (4) hold for expressions, with equality in place of equivalence. The notation for substitutions is extended to allow multiple simultaneous substitutions. If $\vec{v}$ is a vector of distinct variables and $\vec{E}$ is a vector of expressions of the same length as $\vec{v}$, then $P[\vec{v}\backslash\vec{E}]$ stands for P with every occurrence of each variable in $\vec{v}$ simultaneously replaced by the corresponding expression within $\vec{E}$. The above laws can be extended to multiple substitutions.

Substitution on commands is defined in the obvious way, but note that for local variables v and v', $(\circ\vec{x}: [Q])[v\backslash v']$ is defined to be $\circ\vec{x}[v\backslash v']: [Q[v_0, v\backslash v'_0, v']]$.

3. Procedures and parameters

In the real-time context the treatment of parameters is similar to that in non-real-time languages [21]. We provide value and result parameters similar to those of Morgan [28,29]. However, value and result parameter passing mechanisms are not suitable for passing external input and output variables to procedures. For input and output variables, we are interested in not just the initial (value) and final (result) values of parameters, but we are also interested in the value of inputs and outputs over the duration of the execution of the procedure. For this reason, we need to handle external input and output parameters in a different manner to value and result parameters (and the implementation will have to use a technique such as call-by-reference to ensure accesses to inputs and outputs within the procedure body happen immediately). Value and result parameters may be qualified as being auxiliary, e.g., **value aux** $ev : Time_\infty$.

As an example consider the procedure, *Await*. It waits for the Boolean input *sensor* to attain the value of the argument *val* and returns an approximation, pt, to the time that the sensor changes. If the sensor never attains *val*, the procedure never terminates.

procedure *Await*(**input** *sensor* : Boolean; **value** *val* : Boolean; **value aux** $ev : Time_\infty$; **result** pt : *time*)

$$\big\{(sensor \neq val)\ \textbf{over}\ (\tau \ldots ev) \wedge (sensor = val)\ \textbf{over}\ (ev \ldots ev + 10\text{ ms})\big\}; \tag{5}$$

$$\circ pt{:}\ \big[ev_0 = \tau = \infty \vee (ev_0 \leq \tau < \infty \wedge ev_0 \leq pt \leq ev_0 + 10\text{ ms})\big] \tag{6}$$

To allow simpler specification of the procedure, an auxiliary parameter, *ev*, is used; it gives the (future) time of the awaited change.[4] The procedure may assume that *sensor* is not equal to *val* until *ev*, and that once it changes to *val* it will remain equal to *val* for at least 10 ms. The notation

$$(sensor = val)\ \textbf{over}\ (ev \ldots ev + 10\text{ ms}) \tag{7}$$

states that the predicate $(sensor = val)$ holds over the open interval of time from *ev* to $ev + 10$ ms. The input *sensor* is a function from time to Boolean. The notation P **over** S is defined as $(\forall t : S \bullet P\ @\ t)$, where t is a fresh variable.

[4] The context of a call on *Await* needs to guarantee that *ev* has an appropriate value, for example, by initialising the actual parameter using a specification command that has (5) as its postcondition.

Hence (7) is equivalent to

$$\begin{array}{l}(\forall t : (ev \mathinner{..} ev + 10\text{ ms}) \bullet sensor(t) = val)\\ \equiv (\forall t : Time_\infty \bullet ev < t < ev + 10\text{ ms} \Rightarrow sensor(t) = val)\end{array}$$

If the value of *sensor* never changes, i.e., $ev_0 = \infty$, then *Await* never terminates, i.e., $\tau = \infty$. We use open intervals in (5), rather than closed intervals to allow for the cases when the intervals are empty (in the first case when $ev \leq \tau$ and in the second case when ev is infinity). Note that $\infty + 10$ ms is defined to be ∞ and hence the open interval $(ev \mathinner{..} ev + 10\text{ ms})$ is empty when ev is infinity. Also note that within (6) the initial variable ev_0 is used rather than ev because in the case of nontermination there is no final value of the local variables. This ensures that the postcondition (6) satisfies Definition 1 (nontermination state independent) because when $\tau = \infty$, the second disjunct of (6) is false and the first disjunct is equivalent to $ev_0 = \infty$; in this case $\rho.local$ includes ev, val and pt, and Definition 1 (nontermination state independent) requires that $(ev_0 = \infty) \equiv (\forall ev, val, pt \bullet ev_0 = \infty)$ which holds because ev, val and pt do not occur in the predicate.

In an environment, ρ, containing the definition of the procedure *Await*, a Boolean input variable *beam*, a time-valued auxiliary up_t, and a time-valued local variable et, a specification of the form

$$\begin{array}{l}\big\{(beam \neq true)\ \textbf{over}\ (\tau \mathinner{..} up\_t) \wedge (beam = true)\ \textbf{over}\ (up\_t \mathinner{..} up\_t + 10\text{ ms})\big\};\\ \infty et:\ \big[up\_t_0 = \tau = \infty \vee (up\_t_0 \leq \tau < \infty \wedge up\_t_0 \leq et \leq up\_t_0 + 10\text{ ms})\big]\end{array} \tag{8}$$

may be refined to a procedure call: **call** $Await(beam, true, up\_t, et)$. In Section 4 we present a refinement law that allows this refinement to be shown, but first we need to give the semantics of procedure definitions and calls.

Definition 13 (*Procedure Definition*). Consider an environment, ρ, that does not contain p, and a vector of distinct formal parameter declarations, $\vec{d}$. Let ρ' be the environment ρ updated with the formal parameters $\vec{d}$, where the value and result parameters within $\vec{d}$ are declared as local/auxiliary variables within ρ', and let C be a command that is well defined in ρ', and D be a command that is well defined in the environment ρ'', that consists of ρ updated with the definition of procedure p,[5] then a block consisting of the command D in an environment extended with a procedure, p, with body C, has the form |[**procedure** $p(\vec{d})$ $C \bullet D$]| and is defined by

$$EnvOut(|[\ \textbf{procedure}\ p(\vec{d})\ C \bullet D\ ]|)(\rho) = \rho$$
$$\mathcal{M}_\rho\left(|[\ \textbf{procedure}\ p(\vec{d})\ C \bullet D\ ]|\right) \mathrel{\widehat{=}} \mathcal{M}_{\rho''}(D)$$

We develop laws for procedures in the real-time calculus that are similar to those for the non-real-time calculus [28,29]. Wildman et al. [35] present a subset of the rules, but here we give justifications in terms of our semantics. Parameter typing is similar to that for standard programming languages. Hence in this paper we only treat parameter types informally. The following law follows directly from Definition 13 (procedure definition).

Law 14 (*Refine Block Body*). *Given an environment ρ in which the definition* |[**procedure** $p(\vec{d})$ $C \bullet D$]| *is well defined, then if $D \sqsubseteq_{\rho''} D'$, where ρ'' is ρ updated with the definition of p, then*

$$|[\ \textbf{procedure}\ p(\vec{d})\ C \bullet D\ ]| \sqsubseteq_\rho |[\ \textbf{procedure}\ p(\vec{d})\ C \bullet D'\ ]|$$

The following law allows a procedure definition to be introduced. It is a refinement in both directions (denoted $\sqsupseteq_\rho$). It follows from Definition 13 because $\mathcal{M}_{\rho''}(D) = \mathcal{M}_\rho(D)$ as D is well defined in ρ, which does not include p.

Law 15 (*Introduce Procedure*). *Given an environment ρ that does not include p, a vector of distinct formal parameters $\vec{d}$, a command C that is well defined in ρ updated with $\vec{d}$, and a command D that is well defined in ρ, then*

$$D \sqsupseteq_\rho |[\ \textbf{procedure}\ p(\vec{d})\ C \bullet D\ ]|.$$

To define a procedure call we introduce local variables corresponding to all the formal value and result parameters; we use fresh names for the local variables to avoid capturing references to variables in the environment at the point

[5] That is, ρ'' is the same as ρ, except that $\rho''.procs$ has added to it the definition of p as a procedure with body C and formal parameters $\vec{d}$.

of call. For the execution of a call, the values of the actual value parameter expressions are assigned to the local variables corresponding to the formal value parameters, the body of the procedure is executed, and the local variables corresponding to the formal result parameters are assigned to the actual result parameter variables [29]. To allow for the time overheads of the call and return, **idle** commands are included in the definition. Given a vector, $\vec{d}$, of formal parameter declarations, we use the notation $\hat{d}$ to refer to the corresponding vector of names of formal parameters.

Definition 16 (*Procedure Call*). Consider an environment, ρ, in which a procedure, p, is defined as

$$\textbf{procedure}\ p(\vec{d})\ C$$

where $\vec{d}$ is a vector of distinct formal parameters. Let $\vec{i}$, $\vec{o}$, $\vec{v}$ and $\vec{r}$ be the (possibly empty) subvectors of $\hat{d}$ of the names of the inputs, outputs, value and result parameters, respectively. Given

- a vector of inputs, $\vec{ai}$, of the same type as $\vec{i}$;
- a vector of distinct outputs, $\vec{ao}$, of the same type as $\vec{o}$ and that do not occur free in C[6];
- a vector of idle-constant expressions, $\vec{av}$, which is assignment compatible with $\vec{v}$ (for non-auxiliary parameters, the actual parameter expression cannot refer to τ, auxiliaries or inputs);
- a vector of distinct local variables, $\vec{ar}$, with which $\vec{r}$ is assignment compatible (for an auxiliary result formal parameter, the corresponding actual result parameter must be an auxiliary variable); and
- vectors $\vec{v}'$ and $\vec{r}'$ of fresh variables, that are not in the calling environment ρ, and that correspond in number, type and kind (auxiliary or not) with $\vec{v}$ and $\vec{r}$;

then a call on p of the form, $\textbf{call}\ p\left(\hat{d}\left[\frac{\vec{ai},\vec{ao},\vec{av},\vec{ar}}{\vec{i},\vec{o},\vec{v},\vec{r}}\right]\right)$, is well defined and is equivalent to

$$\left|\left[\ \textbf{var}\,\vec{v}';\ \textbf{var}\,\vec{r}';\ \textbf{idle};\ \vec{v}' := \vec{av};\ C\left[\frac{\vec{ai},\vec{ao},\vec{v}',\vec{r}'}{\vec{i},\vec{o},\vec{v},\vec{r}}\right];\ \vec{ar} := \vec{r}';\ \textbf{idle}\ \right]\right|.$$

Scoping of variables. Note that the environment at the point of call may be an extension of that at the point of declaration. Because primitive commands all have a frame and leave variables outside the frame unchanged, the effect of executing the call in the extended environment is to leave the additional variables in the calling environment unchanged. Consider the following program which declares a variable x, defines a procedure p in terms of x, and then enters a new scope that redeclares x.

```
|[ var x : T1;
    procedure p() ∞x: [Q];
    |[ var x : T2;
        ∞x: [Q];   -- cannot be refined to a call to p
        ...
    ]|
]|
```

Note that, although the specification command within the nested block is syntactically identical to the body of the procedure p, the specification command cannot be refined to p because the variable x used in the definition of p is statically bound to the first x (of type T_1), whereas the x referred to in the specification command is the second x (of type T_2). We avoid this type of problem by disallowing the redeclaration of a variable within a nested scope, i.e., the above program is not valid. Our definition of local variable blocks has already banned such redeclarations. This issue with scoping already arises in the standard refinement calculus. In a language with nested scoping, if we want to allow refinement of a specification command to a procedure call, then the global variables used in the specification have to be the same as those used in the procedure definition. The simplest way to avoid confusion is to disallow redeclaration of a variable within a nested scope. Moreover, using the same name for two different variables within the same overall

[6] An exception may be made if a name in $\vec{ao}$ and the corresponding name in $\vec{o}$ are the same.

scope is not good programming style. For the same reasons we do not allow redefinition of a procedure within a nested scope. However, we do allow reuse of names for procedure parameters.

We have chosen to treat nested scoping of procedure definitions in this paper because it is more complex than simple global scoping as found in programming languages such as C and scoping associated with classes as found in object-oriented languages but the rules presented here can be adapted to those contexts.

Refining the body of a procedure refines the program in which it occurs. Groves [6], Staples [34], and Wildman et al. [35] all make use of procedure definitions that include both an interface specification, S, and an implementation, I, of a procedure: $|[\ \textbf{procedure}\ p(\vec{d})\ S \sqsubseteq I \bullet D\ ]|$. With this form of definition only the implementation I can be refined. Here we have used the simpler form of definition as used by Morgan [28] but the rules can easily be adapted to the more general form.

Law 17 (*Refine Procedure Body*). *Given an environment ρ in which the block $|[\ \textbf{procedure}\ p(\vec{d})\ C \bullet D\ ]|$ is well defined, then if $C \sqsubseteq_{\rho'} C'$, where ρ' is ρ updated with the declarations $\vec{d}$, then*

$$|[\ \textbf{procedure}\ p(\vec{d})\ C \bullet D\ ]| \sqsubseteq_{\rho} |[\ \textbf{procedure}\ p(\vec{d})\ C' \bullet D\ ]|.$$

Proof. From the definition of refinement (see Fig. 1) we need to show

$$\begin{array}{l} \mathcal{M}_\rho\left(|[\ \textbf{procedure}\ p(\vec{d})C' \bullet D\ ]|\right) \Rrightarrow \mathcal{M}_\rho\left(|[\ \textbf{procedure}\ p(\vec{d})C \bullet D\ ]|\right) \\ \equiv \text{Definition 13 (procedure definition)} \\ \mathcal{M}_{\rho''}(D) \Rrightarrow \mathcal{M}_{\rho'}(D) \end{array}$$

where ρ' is ρ updated with the first definition of p (with body C), and ρ'' is ρ updated with the second definition of p (with body C'). This can be shown by structural induction over the form of D. The only case of interest is a procedure call on p, and from Definition 16 (procedure call) we need to show

$$\left|\left[\ \textbf{var}\,\vec{v}'; \textbf{var}\,\vec{r}'; \textbf{idle}; \vec{v}' := \vec{a}v; C\left[\frac{\vec{a}i, \vec{a}o, \vec{v}', \vec{r}'}{\vec{i}, \vec{o}, \vec{v}, \vec{r}}\right]; \vec{a}r := \vec{r}'; \textbf{idle}\ \right]\right|$$

$$\sqsubseteq_\rho \left|\left[\ \textbf{var}\,\vec{v}'; \textbf{var}\,\vec{r}'; \textbf{idle}; \vec{v}' := \vec{a}v; C'\left[\frac{\vec{a}i, \vec{a}o, \vec{v}', \vec{r}'}{\vec{i}, \vec{o}, \vec{v}, \vec{r}}\right]; \vec{a}r := \vec{r}'; \textbf{idle}\ \right]\right|$$

which follows because C is refined by C' and $\vec{a}o$ does not occur free in C or C' and $\vec{v}'$ and $\vec{r}'$ are fresh vectors of variables. This last step can be proved by reducing C and C' to their equivalent specification commands and then showing that the renaming preserves refinement.

4. Laws for introducing procedure calls

In Section 4.4 we present a general law that handles procedures with an arbitrary number of parameters. In order to understand the law and its proof, we begin by examining parameterless procedures (Section 4.1) and then examine procedures with value parameters (Section 4.2), and procedures with result parameters (Section 4.3).

4.1. Parameterless procedure calls

Consider a parameterless procedure, p, defined as follows.

$$\textbf{procedure}\ p()\ \ \{P\}; \infty\vec{x}: [Q]$$

We would like to develop a refinement law of the form

$$\{P\}; \infty\vec{x}: [Q] \sqsubseteq_\rho \textbf{call}\ p() \tag{9}$$

From Definition 16 (procedure call)

$$\textbf{call}\ p() \sqsupseteq_\rho \textbf{idle}; \{P\}; \infty\vec{x}: [Q]; \textbf{idle} \tag{10}$$

where the **idle** commands allow for the time overheads of procedure entry and exit. In this case there are no value or result parameters to declare and assign. In order to show the refinement (9), we need to show

$$\{P\}; \infty\vec{x}: [Q] \sqsubseteq_\rho \textbf{idle}; \{P\}; \infty\vec{x}: [Q]; \textbf{idle} \tag{11}$$

Unfortunately, this does not hold in general as P and Q may be time dependent, e.g., they may refer to the current time variable τ. On the left side of the refinement, P holds initially, but on the right side this corresponds to the instant before the execution of the first **idle** command. However, because the **idle** command may take time to execute, P may no longer hold on termination of the **idle** command. To avoid this problem, we insist that the precondition P is *idle-invariant*. Note that *idle-constant* differs from *idle-invariant* because the former requires equivalence, whereas the latter only requires implication.

Definition 18 (*Idle-invariant*). A single-state predicate P over an environment ρ is *idle invariant* provided,

$$\tau_0 \leq \tau < \infty \wedge const(\rho.out, [\tau_0 \mathinner{..} \tau]) \wedge P\,[\tau \backslash \tau_0] \Rrightarrow P.$$

This states that if the values of the program outputs do not change between times τ_0 and τ and the values of the local and auxiliary variables are unchanged (in the above formula this is implicit in that their occurrences in P and $P\,[\tau \backslash \tau_0]$ refer to the same name), then if P holds at τ_0, it also holds at τ. For example, the precondition (5) of the procedure *Await* is idle invariant:

$$\begin{aligned} &\tau_0 \leq \tau < \infty \wedge const(\rho.out, [\tau_0 \mathinner{..} \tau]) \wedge \\ &(sensor \neq val)\ \textbf{over}\ (\tau_0 \mathinner{..} ev) \wedge (sensor = val)\ \textbf{over}\ (ev \mathinner{..} ev + 10\ \text{ms}) \\ \Rrightarrow\ &(sensor \neq val)\ \textbf{over}\ (\tau \mathinner{..} ev) \wedge (sensor = val)\ \textbf{over}\ (ev \mathinner{..} ev + 10\ \text{ms}) \end{aligned}$$

Because $\tau_0 \leq \tau$, it follows that $(\tau \mathinner{..} ev) \subseteq (\tau_0 \mathinner{..} ev)$ and hence any predicate that holds over the latter interval must also hold over any interval it contains (including holding vacuously over the empty interval if $\tau > ev$).

Theorem 19. *If τ does not occur free in P, then P is idle invariant.*

The definition of idle-invariant above has been chosen to be the most general such that the following law holds.

Law 20 (*Idle-invariant Assumption*). *If P is an idle-invariant single-state predicate over an environment ρ, then*

$$\{P\}; \textbf{idle} \sqsubseteq_\rho \textbf{idle}; \{P\}$$

Proof. In the proof the single-state predicate P is used as a postcondition, where it is interpreted as the equivalent relation (that does not refer to zero-subscripted variables). The second use of Law 5 (strengthen postcondition) below replaces the postcondition P with the (weaker) postcondition *true*, which is equivalent to P in the context of the assumption P, given that P is idle invariant.

$\{P\}$; **idle**
$\sqsupseteq_\rho$ definition of **idle**
$\{P\}; \varnothing: [true]$
$\sqsubseteq_\rho$ Law 5 (strengthen postcondition)
$\{P\}; \varnothing: [P]$
$\sqsupseteq_\rho$ Law 9 (post assumption) as P is single state
$\{P\}; \varnothing: [P]; \{P\}$
$\sqsubseteq_\rho$ Law 5 (strengthen postcondition) as P is idle invariant
$\{P\}; \varnothing: [true]; \{P\}$
$\sqsubseteq_\rho$ Law 8 (remove assumption); definition of **idle**
idle; $\{P\}$

If during a refinement we have an assumption $\{P\}$ that is not idle invariant we can convert it to an idle-invariant assumption by introducing an auxiliary time variable t and assigning it the current time. The assumption can be replaced by "$t := \tau; \{P\,[\tau \backslash t]\}$", and $P\,[\tau \backslash t]$ is idle invariant.

Returning to the refinement (11), the postcondition, Q, of the specification command may be time dependent. Because Q may reference both τ_0 and τ we require that it is invariant to increases in τ_0 and τ provided none of

the other variables under the control of the program change. To represent these properties we use the terms *pre-idle-invariant* and *post-idle-invariant*.

Definition 21 (*Pre-idle-invariant*). Given a single-state predicate, P, over an environment, ρ, a relation Q over ρ is *pre-idle-invariant* in context P provided, for a fresh variable τ',

$$\tau' < \infty \wedge \tau_0 \leq \tau' \leq \tau \wedge const(\rho.out, [\tau_0 \mathinner{..} \tau']) \wedge P_0 \wedge Q\left[\tau_0 \backslash \tau'\right] \Rrightarrow Q$$

where P_0 stands for P with τ and local variables replaced by their zero-subscripted counterparts.

Theorem 22. *If τ_0 does not occur free in Q, then Q is pre-idle-invariant in any context.*

This theorem can be used to show that postcondition (6) of procedure *Await* is pre-idle-invariant, because (6) does not contain free occurrences of τ_0. The definition of pre-idle-invariant above has been chosen to be the most general such that the following law holds.

Law 23 (*Pre-idle-invariant Postcondition*). *Given a single-state predicate, P over an environment, ρ, if a relation Q over ρ is pre-idle-invariant in context P, then for any frame over ρ containing locals $\vec{x}$, and outputs $\vec{o}$,*

$$\{P\}; \infty\vec{x}, \vec{o}: [Q] \sqsubseteq_\rho \{P\}; \mathbf{idle}; \infty\vec{x}, \vec{o}: [Q].$$

Proof.

$\{P\}; \infty\vec{x}, \vec{o}: [Q]$
$\sqsubseteq_\rho$ Law 10 (sequential); see below
$\{P\}; \infty\vec{x}, \vec{o}: [\tau < \infty \wedge const(\vec{o}, [\tau_0 \mathinner{..} \tau]) \wedge \vec{x} = \vec{x}_0]; \infty\vec{x}, \vec{o}: [Q]$
$\sqsupseteq_\rho$ Law 6 (contract frame)
$\{P\}; \infty\varnothing: [\tau < \infty]; \infty\vec{x}, \vec{o}: [Q]$
$\sqsupseteq_\rho$ definition of **idle**
$\{P\}; \mathbf{idle}; \infty\vec{x}, \vec{o}: [Q]$

For the step involving the sequential composition we have the following side condition. Let $\vec{no}$ be the outputs in $\rho.out$ excluding $\vec{o}$, and $\vec{nx}$ be the variables in $\rho.var \cup \rho.aux$ excluding $\vec{x}$.

$$\left(\begin{array}{l}\tau_0 < \infty \wedge P_0 \wedge const(\vec{no}, [\tau_0 \mathinner{..} \tau]) \wedge (\tau < \infty \Rightarrow \vec{nx}_0 = \vec{nx}) \wedge \\ \left(\exists \tau', \vec{x}' \bullet \tau_0 \leq \tau' \leq \tau \wedge \tau' < \infty \wedge const(\vec{o}, [\tau_0 \mathinner{..} \tau']) \wedge \vec{x}' = \vec{x}_0 \wedge Q\left[\tau_0, \vec{x}_0 \backslash \tau', \vec{x}'\right]\right)\end{array}\right) \Rrightarrow Q$$

The existential quantification over $\vec{x}'$ can be eliminated using the one-point rule because $\vec{x}' = \vec{x}_0$, and the quantification over τ' can be converted into a universal quantification using the law that $(\exists \tau' \bullet R_1) \Rrightarrow R_2$ is equivalent to $\forall \tau' \bullet (R_1 \Rrightarrow R_2)$, provided τ' does not occur free in R_2.

$$\forall \tau' \bullet \left(\left(\begin{array}{l}\tau_0 < \infty \wedge P_0 \wedge const(\vec{no}, [\tau_0 \mathinner{..} \tau]) \wedge (\tau < \infty \Rightarrow \vec{nx}_0 = \vec{nx}) \wedge \\ \tau_0 \leq \tau' \leq \tau \wedge \tau' < \infty \wedge const(\vec{o}, [\tau_0 \mathinner{..} \tau']) \wedge Q\left[\tau_0 \backslash \tau'\right]\end{array}\right) \Rrightarrow Q\right)$$

This follows from the fact that Q is pre-idle-invariant and $\vec{o}$ union $\vec{no}$ contains all the output variables ($\rho.out$).

Definition 24 (*Post-idle-invariant*). Given a single-state predicate, P over an environment, ρ, a relation Q over ρ is *post-idle-invariant* in context P provided, for a fresh variable τ',

$$\tau_0 \leq \tau' \leq \tau < \infty \wedge const(\rho.out, [\tau' \mathinner{..} \tau]) \wedge P_0 \wedge Q\left[\tau \backslash \tau'\right] \Rrightarrow Q$$

where P_0 stands for P with τ and local variables replaced by their zero-subscripted counterparts.

The postcondition (6) of procedure *Await* is post-idle-invariant:

$$\begin{array}{l}\tau_0 \leq \tau' \leq \tau < \infty \wedge const(\rho.out, [\tau' \mathinner{..} \tau]) \wedge \\ (ev_0 = \tau' = \infty \vee (ev_0 \leq \tau' < \infty \wedge ev_0 \leq pt \leq ev_0 + 10\text{ ms})) \\ \Rrightarrow (ev_0 = \tau = \infty \vee (ev_0 \leq \tau < \infty \wedge ev_0 \leq pt \leq ev_0 + 10\text{ ms}))\end{array}$$

Because τ' and τ are both less than infinity, the disjuncts $ev_0 = \tau' = \infty$ and $ev_0 = \tau = \infty$ are both false, and the implication follows because $\tau' \leq \tau$.

Theorem 25. *If τ does not occur free in Q, then Q is post-idle-invariant in any context.*

The definition of post-idle-invariant above has been chosen to be the most general such that the following law holds.

Law 26 (*Post-idle-invariant Postcondition*). *If a relation Q over an environment ρ is post-idle-invariant in context P, then for any frame over ρ with locals $\vec{x}$ and outputs $\vec{o}$,*

$$\{P\}; \infty\vec{x}, \vec{o}: [Q] \sqsubseteq_\rho \{P\}; \infty\vec{x}, \vec{o}: [Q]; \mathbf{idle}.$$

Proof.

$\{P\}; \infty\vec{x}, \vec{o}: [Q]$
$\sqsubseteq_\rho$ Law 10 (sequential); see below
$\quad \{P\}; \infty\vec{x}, \vec{o}: [Q]; \infty\vec{x}, \vec{o}: [\tau < \infty \wedge const(\vec{o}, [\tau_0 \ldots \tau]) \wedge \vec{x} = \vec{x}_0]$
$\sqsubseteq_\rho$ Law 6 (contract frame)
$\quad \{P\}; \infty\vec{x}, \vec{o}: [Q]; \infty\varnothing: [\tau < \infty]$
$\sqsubseteq_\rho$ definition of **idle**
$\quad \{P\}; \infty\vec{x}, \vec{o}: [Q]; \mathbf{idle}$

For the step involving the sequential composition we have the following side condition. Let $\vec{no}$ be the outputs in $\rho.out$ excluding $\vec{o}$, and $\vec{nx}$ be the variables in $\rho.var \cup \rho.aux$ excluding $\vec{x}$.

$$\left(\begin{array}{l}\tau_0 < \infty \wedge P_0 \wedge const(\vec{no}, [\tau_0 \ldots \tau]) \wedge (\tau < \infty \Rightarrow \vec{nx}_0 = \vec{nx}) \wedge \\ (\exists \tau', \vec{x}' \bullet \tau_0 \le \tau' \le \tau \wedge Q\left[\tau, \vec{x} \backslash \tau', \vec{x}'\right] \wedge (\tau' < \infty \Rightarrow \tau < \infty \wedge const(\vec{o}, [\tau' \ldots \tau]) \wedge \vec{x} = \vec{x}'))\end{array}\right) \Rrightarrow Q$$

For the case when $\tau' < \infty$, the existential quantification over $\vec{x}'$ can be eliminated using the one-point rule and the quantification over τ' can be converted to a universal quantification as in Law 23 (pre-idle-invariant postcondition).

$$\forall \tau' \bullet \left(\left(\begin{array}{l}\tau_0 < \infty \wedge P_0 \wedge const(\vec{no}, [\tau_0 \ldots \tau]) \ \wedge (\tau < \infty \Rightarrow \vec{nx}_0 = \vec{nx}) \wedge \\ \tau_0 \le \tau' \le \tau < \infty \ \wedge const(\vec{o}, [\tau' \ldots \tau]) \wedge Q\left[\tau \backslash \tau'\right]\end{array}\right) \Rrightarrow Q\right)$$

This follows from the fact that Q is post-idle-invariant. For the case when $\tau' = \infty$, it follows that $\tau' = \tau$ and $\tau = \infty$, and hence $Q\left[\tau, \vec{x} \backslash \tau', \vec{x}'\right] \equiv Q\left[\vec{x} \backslash \vec{x}'\right]$, but as Q is the postcondition of a specification command, it satisfies Definition 1 (nontermination state independent) and hence when $\tau = \infty$, $Q\left[\vec{x} \backslash \vec{x}'\right] \equiv Q$.

Law 27 (*Idle-invariant Specification*). *Given an environment, ρ, in which P is an idle-invariant, single-state predicate, Q is a pre- and post-idle-invariant relation in context P, and $\vec{x}$ is a vector of variables over ρ, that does not include inputs, then*

$$\{P\}; \infty\vec{x}: [Q] \sqsubseteq_\rho \mathbf{idle}; \{P\}; \infty\vec{x}: [Q]; \mathbf{idle}.$$

Proof.

$\{P\}; \infty\vec{x}: [Q]$
$\sqsubseteq_\rho$ by Law 26 (post-idle-invariant postcondition)
$\quad \{P\}; \infty\vec{x}: [Q]; \mathbf{idle}$
$\sqsubseteq_\rho$ by Law 23 (pre-idle-invariant postcondition)
$\quad \{P\}; \mathbf{idle}; \infty\vec{x}: [Q]; \mathbf{idle}$
$\sqsubseteq_\rho$ by Law 20 (idle-invariant assumption)
$\quad \mathbf{idle}; \{P\}; \infty\vec{x}: [Q]; \mathbf{idle}$

The following law follows from Law 27 (idle-invariant specification) and (10), which follows from Definition 16 (procedure call).

Law 28 (*Parameterless Procedure Call*). *Given an environment, ρ, in which p is defined by*

$$\mathbf{procedure}\ p()\ \ \{P\}; \infty\vec{x}: [Q],$$

where P is an idle-invariant, single-state predicate over ρ, Q is a pre- and post-idle-invariant relation over ρ in context P, and $\vec{x}$ is a vector of variables over ρ, that does not include inputs, then

$$\{P\}; \infty\vec{x}: [Q] \sqsubseteq_\rho \mathbf{call}\ p().$$

4.2. Value parameters

Consider a procedure pv with a vector, $\vec{v}$, of value parameters, that may include auxiliary value parameters. We use the notation **value** $\vec{v} : \vec{T}$ to stand for the sequence of value parameter declarations where each element of $\vec{v}$ is of type the corresponding element of $\vec{T}$, where $\vec{T}$ is a vector of the same length as $\vec{v}$ of nonempty types. The global variables that can be modified by the procedure are represented by a vector $\vec{x}$, whose names are disjoint from $\vec{v}$.

$$\textbf{procedure}\ pv(\textbf{value}\,\vec{v} : \vec{T})\ \ \{P\};\ \infty\vec{v}, \vec{x}{:}\,[Q] \tag{12}$$

In a calling environment ρ, that does not contain $\vec{v}'$, from Definition 16 (procedure call), a call on pv with idle-constant actual parameter expressions $\vec{av}$ of type $\vec{T}$ satisfies the following.

$$\textbf{call}\ pv(\vec{av}) \sqsubseteq_\rho \left| \left[\ \textbf{var}\,\vec{v}' : \vec{T};\ \textbf{idle};\ \vec{v}' := \vec{av};\ \left\{ P\left[\begin{matrix}\vec{v}'\\ \vec{v}\end{matrix}\right]\right\};\ \infty\vec{v}', \vec{x}{:}\ \ \left[Q\left[\begin{matrix}\vec{v}'_0, \vec{v}'\\ \vec{v}_0, \vec{v}\end{matrix}\right]\right];\ \textbf{idle}\ \right]\right| \tag{13}$$

This introduces a local variable block for the variables representing the formal value parameters. Hence we need laws for reasoning about local variable blocks. If a specification is idle invariant then a local variable block can be introduced around it. The following law follows from Law 27 (idle-invariant specification) and Law 12 (local and auxiliary variables).

Law 29 (*Local Variables Idle-invariant*). *In an environment, ρ, if P is a idle-invariant single-state predicate, Q is a pre- and post-idle invariant relation in context P, and $\vec{x}$ is a vector of noninput variables, then for variables $\vec{w}$ not in ρ,*

$$\{P\};\ \infty\vec{x}{:}\,[Q] \sqsubseteq |[\ \textbf{var}\,\vec{w} : \vec{T};\ \{P\};\ \infty\vec{w}, \vec{x}{:}\,[Q]\ ]|$$

A variant of Law 29 (local variables idle-invariant) is to allow initialisation of the local variable.

Law 30 (*Init Variables Idle-invariant*). *In an environment, ρ, if P is an idle-invariant single-state predicate, Q is a pre- and post-idle-invariant relation in context P, $\vec{x}$ is a vector of noninput variables, $\vec{E}$ is a vector of idle-constant expressions, then for fresh variables $\vec{w}$ (i.e., not in ρ) that the type of $\vec{E}$ is assignment compatible with,*

$$\begin{array}{l} \quad \{P \wedge def(\vec{E} \mathbin{@} \tau)\};\ \infty\vec{x}{:}\,[Q] \\ \sqsubseteq_\rho\ |[\ \textbf{var}\,\vec{w} : \vec{T};\ \vec{w} := \vec{E};\ \{P \wedge \vec{w} = \vec{E} \mathbin{@} \tau\};\ \infty\vec{w}, \vec{x}{:}\,[Q]\ ]| \end{array}$$

Proof. Because P is idle invariant and $\vec{E}$ is idle-constant, $P \wedge def(\vec{E} \mathbin{@} \tau)$ is also idle invariant.

$$\begin{array}{l} \quad \{P \wedge def(\vec{E} \mathbin{@} \tau)\};\ \infty\vec{x}{:}\,[Q] \\ \sqsubseteq_\rho \text{ by Law 29 (local variables idle-invariant)} \\ \quad |[\ \textbf{var}\,\vec{w} : \vec{T};\ \{P \wedge def(\vec{E} \mathbin{@} \tau)\};\ \infty\vec{w}, \vec{x}{:}\,[Q]\ ]| \\ \sqsubseteq_\rho \text{ by Law 10 (sequential); } \vec{w}_0 \text{ does not occur in } Q \\ \quad |[\ \textbf{var}\,\vec{w} : \vec{T};\ \{P \wedge def(\vec{E} \mathbin{@} \tau)\};\ \vec{w}{:}\,[P \wedge \vec{w} = \vec{E} \mathbin{@} \tau];\ \infty\vec{w}, \vec{x}{:}\,[Q]\ ]| \\ \sqsubseteq_\rho \text{ by Law 9 (post assumption)} \\ \quad |[\ \textbf{var}\,\vec{w} : \vec{T};\ \{P \wedge def(\vec{E} \mathbin{@} \tau)\};\ \vec{w}{:}\,[P \wedge \vec{w} = \vec{E} \mathbin{@} \tau];\ \{P \wedge \vec{w} = \vec{E} \mathbin{@} \tau\};\ \infty\vec{w}, \vec{x}{:}\,[Q]\ ]| \\ \sqsubseteq_\rho \text{ by Law 5 (strengthen postcondition); } P \text{ idle-invariant and no free } \vec{w} \\ \quad |[\ \textbf{var}\,\vec{w} : \vec{T};\ \{P \wedge def(\vec{E} \mathbin{@} \tau)\};\ \vec{w}{:}\,[\vec{w} = \vec{E}_0 \mathbin{@} \tau_0];\ \{P \wedge \vec{w} = \vec{E} \mathbin{@} \tau\};\ \infty\vec{w}, \vec{x}{:}\,[Q]\ ]| \\ \sqsubseteq_\rho \text{ by the definition of assignment} \\ \quad |[\ \textbf{var}\,\vec{w} : \vec{T};\ \{P \wedge def(\vec{E} \mathbin{@} \tau)\};\ \vec{w} := \vec{E};\ \{P \wedge \vec{w} = \vec{E} \mathbin{@} \tau\};\ \infty\vec{w}, \vec{x}{:}\,[Q]\ ]| \\ \sqsubseteq_\rho \text{ by Law 8 (remove assumption)} \\ \quad |[\ \textbf{var}\,\vec{w} : \vec{T};\ \vec{w} := \vec{E};\ \{P \wedge \vec{w} = \vec{E} \mathbin{@} \tau\};\ \infty\vec{w}, \vec{x}{:}\,[Q]\ ]| \end{array}$$

We would like to develop a law that allows a specification command to be refined to a call on procedure pv. The precondition, P, of the procedure body is required to hold for the formal parameters $\vec{v}$ on entry to the variable block (13). For the call this means the precondition should hold for the actual parameter expressions $\vec{av}$, i.e., $P[\vec{v}\backslash\vec{av} \mathbin{@} \tau]$ should hold initially. In addition, due to the procedure entry time overheads, this precondition should be idle invariant so that it is still true after procedure entry. This holds if P is idle invariant and the actual parameters $\vec{av}$ are idle-constant.

Theorem 31. *Given an environment, ρ, if P is an idle-invariant predicate and $\vec{av}$ are idle-constant expressions, then $P[\vec{v}\backslash\vec{av} \mathbin{@} \tau]$ is also idle invariant.*

Proof. If we assume $\tau_0 \leq \tau < \infty \wedge const(\rho.out, [\tau_0 \mathinner{..} \tau])$, then by Definition 18 (idle-invariant), $P[\tau\backslash\tau_0] \Rrightarrow P$, and by Definition 3 (idle-constant) $\vec{av} \mathbin{@} \tau_0 = \vec{av} \mathbin{@} \tau$. We need to show $P[\vec{v}\backslash\vec{av} \mathbin{@} \tau][\tau\backslash\tau_0] \Rrightarrow P[\vec{v}\backslash\vec{av} \mathbin{@} \tau]$ under the same assumptions.

$$
\begin{array}{ll}
 & P\left[\vec{v}\backslash\vec{av} \mathbin{@} \tau\right][\tau\backslash\tau_0] \\
\equiv & \text{by (3) as } \vec{v} \text{ and } \tau_0 \text{ are distinct variables} \\
 & P[\tau\backslash\tau_0]\left[\vec{v}\backslash(\vec{av} \mathbin{@} \tau)[\tau\backslash\tau_0]\right] \\
\equiv & \text{as } \vec{av} \text{ are idle-constant } (\vec{av} \mathbin{@} \tau)[\tau\backslash\tau_0] = \vec{av} \mathbin{@} \tau_0 = \vec{av} \mathbin{@} \tau \\
 & P[\tau\backslash\tau_0]\left[\vec{v}\backslash\vec{av} \mathbin{@} \tau\right] \\
\Rrightarrow & \text{by (4) from } P[\tau\backslash\tau_0] \Rrightarrow P \\
 & P\left[\vec{v}\backslash\vec{av} \mathbin{@} \tau\right].
\end{array}
$$

The following corollary follows using Theorem 4.

Corollary 32. *Given an environment, ρ, if τ does not occur free in $\vec{av}$ and $\vec{av}$ contains no references to inputs, then if P is idle invariant, then so is $P[\vec{v}\backslash\vec{av} \mathbin{@} \tau]$.*

Within the procedure the value parameters, $\vec{v}$, can be used as local variables, and they may appear in the postcondition, Q, of the body of the procedure. However, $\vec{v}$ are local to the procedure and not returned by it, and hence within the specification being refined to a call, all that can be deduced in the postcondition is $(\exists \vec{v} : \vec{T} \bullet Q)$. Often Q contains no references to $\vec{v}$ and the existential quantifier can be eliminated. Q typically contain references to the initial value of $\vec{v}$, i.e., $\vec{v}_0$. In the specification being refined these correspond to the initial value of the actual parameter expressions, $\vec{av}_0 \mathbin{@} \tau_0$, where $\vec{av}_0$ stands for the expression $\vec{av}$ with local variables replaced by their zero-subscripted counterparts. The postcondition becomes $(\exists \vec{v} : \vec{T} \bullet Q[\vec{v}_0\backslash\vec{av}_0 \mathbin{@} \tau_0])$. In addition, due to the procedure entry and exit overheads, this postcondition should be both pre- and post-idle-invariant. This holds if Q is both pre- and post-idle-invariant, and the actual parameters $\vec{av}$ are idle-constant. The following theorem has a proof similar to that of Theorem 31.

Theorem 33. *Given an environment, ρ, if Q is a pre- and post-idle-invariant relation in context P and $\vec{av}$ are idle-constant expressions, then $Q[\vec{v}_0\backslash\vec{av}_0 \mathbin{@} \tau_0]$ is a pre- and post-idle-invariant relation in context $P[\vec{v}\backslash\vec{av} \mathbin{@} \tau]$.*

The following law refines a specification to the local variable block (13), which is equivalent to **call** $pv(\vec{av})$. The law does not include the refinement to the call so that it can be reused in the general law in Section 4.4.

Law 34 (*Value Parameters*). *Given an environment, ρ, in which P is an idle-invariant single-state predicate, Q is a pre- and post-idle-invariant relation in context P, and $\vec{av}$ are idle-constant expressions of type $\vec{T}$ (elements of $\vec{av}$ corresponding to non-auxiliary variables in $\vec{v}$ must not contain references to τ, auxiliaries or inputs), then*

$$
\begin{array}{ll}
 & \left\{ P\left[\dfrac{\vec{av} \mathbin{@} \tau}{\vec{v}}\right] \wedge def(\vec{av} \mathbin{@} \tau) \right\}; \infty\vec{x}: \left[\left(\exists \vec{v} : \vec{T} \bullet Q\left[\dfrac{\vec{av}_0 \mathbin{@} \tau_0}{\vec{v}_0}\right]\right)\right] \\
\sqsubseteq_\rho & \left|\left[\mathbf{var}\, \vec{v}' : \vec{T}; \mathbf{idle}; \vec{v}' := \vec{av}; \left\{ P\left[\dfrac{\vec{v}'}{\vec{v}}\right] \right\}; \infty\vec{v}', \vec{x}: \left[Q\left[\dfrac{\vec{v}'_0, \vec{v}'}{\vec{v}_0, \vec{v}}\right] \right] ; \mathbf{idle} \right]\right|
\end{array}
$$

Proof. In the second step of the proof, the postcondition is strengthened by removing the existential quantification over $\vec{v}'$ (given that $\vec{v}'$ are in scope), and $\vec{av}_0 \mathbin{@} \tau_0$ is replaced by $\vec{v}'_0$ because the assumption implies $\vec{v}' = \vec{av} \mathbin{@} \tau$ holds immediately before the specification command.

$$
\begin{array}{ll}
 & \left\{ P\left[\dfrac{\vec{av} \mathbin{@} \tau}{\vec{v}}\right] \wedge def(\vec{av} \mathbin{@} \tau) \right\}; \infty\vec{x}: \left[\left(\exists \vec{v} : \vec{T} \bullet Q\left[\dfrac{\vec{av}_0 \mathbin{@} \tau_0}{\vec{v}_0}\right]\right)\right] \\
\sqsubseteq_\rho & \text{by Law 30 (init variables idle-invariant); rename bound variables to } \vec{v}' \\
 & \left|\left[\mathbf{var}\, \vec{v}' : \vec{T}; \vec{v}' := \vec{av}; \left\{ P\left[\dfrac{\vec{av} \mathbin{@} \tau}{\vec{v}}\right] \wedge \vec{v}' = \vec{av} \mathbin{@} \tau \right\}; \infty\vec{v}', \vec{x}: \left[\left(\exists \vec{v}' : T \bullet Q\left[\dfrac{\vec{av}_0 \mathbin{@} \tau_0, \vec{v}'}{\vec{v}_0, \vec{v}}\right]\right)\right] \right]\right|
\end{array}
$$

$\sqsubseteq_\rho$ by Law 5 (strengthen postcondition) given $\vec{v}'_0 = \vec{a}v_0 \mathbin{@} \tau_0$

$$\left|\left[\mathbf{var}\, \vec{v}' : \vec{T}; \vec{v}' := \vec{a}v; \left\{ P \left[\frac{\vec{a}v \mathbin{@} \tau}{\vec{v}} \right] \wedge \vec{v}' = \vec{a}v \mathbin{@} \tau \right\}; \infty \vec{v}', \vec{x}: \left[Q \left[\frac{\vec{v}'_0, \vec{v}'}{\vec{v}_0, \vec{v}} \right] \right] \right]\right|$$

$\sqsubseteq_\rho$ by Law 7 (weaken assumption)

$$\left|\left[\mathbf{var}\, \vec{v}' : \vec{T}; \vec{v}' := \vec{a}v; \left\{ P \left[\frac{\vec{v}'}{\vec{v}} \right] \right\}; \infty \vec{v}', \vec{x}: \left[Q \left[\frac{\vec{v}'_0, \vec{v}'}{\vec{v}_0, \vec{v}} \right] \right] \right]\right|$$

$\sqsubseteq_\rho$ by Law 27 (idle-invariant specification)

$$\left|\left[\mathbf{var}\, \vec{v}' : \vec{T}; \vec{v}' := \vec{a}v; \mathbf{idle}; \left\{ P \left[\frac{\vec{v}'}{\vec{v}} \right] \right\}; \infty \vec{v}', \vec{x}: \left[Q \left[\frac{\vec{v}'_0, \vec{v}'}{\vec{v}_0, \vec{v}} \right] \right]; \mathbf{idle} \right]\right|$$

$\sqsubseteq_\rho$ as $\vec{a}v$ is idle-constant and $\vec{v}'$ local variables

$$\left|\left[\mathbf{var}\, \vec{v}' : \vec{T}; \mathbf{idle}; \vec{v}' := \vec{a}v; \left\{ P \left[\frac{\vec{v}'}{\vec{v}} \right] \right\}; \infty \vec{v}', \vec{x}: \left[Q \left[\frac{\vec{v}'_0, \vec{v}'}{\vec{v}_0, \vec{v}} \right] \right]; \mathbf{idle} \right]\right|$$

A special case of the above law occurs if $\vec{v}$ are not modified and they do not occur free in Q, in which case the existential quantifier can be eliminated and $\vec{v}'$ can be removed from the frame (using Law 6 (contract frame)).

4.3. Result parameters

We consider a procedure pr with a vector, $\vec{r}$, of distinct result parameters, that is disjoint from $\vec{x}$, and may include auxiliary result parameters.

$$\mathbf{procedure}\ pr(\mathbf{result}\, \vec{r} : \vec{T})\ \ \{P\}; \infty \vec{r}, \vec{x}: [Q] \tag{14}$$

In a calling environment ρ that does not contain $\vec{r}'$, from Definition 16 a call on pr with distinct actual parameter variables $\vec{a}r$ of type $\vec{T}$ satisfies

$$\mathbf{call}\ pr(\vec{a}r) \sqsupseteq_\rho \left|\left[\mathbf{var}\, \vec{r}' : \vec{T}; \mathbf{idle}; \left\{ P \left[\frac{\vec{r}'}{\vec{r}} \right] \right\}; \infty \vec{r}', \vec{x}: \left[Q \left[\frac{\vec{r}'_0, \vec{r}'}{\vec{r}_0, \vec{r}} \right] \right]; \vec{a}r := \vec{r}'; \mathbf{idle} \right]\right|.$$

We would like to develop a law to refine a specification to a call on pr. Within the procedure, local variables $\vec{r}'$ are declared and their final value is assigned to the actual parameters $\vec{a}r$. Because $\vec{r}'$ are local to the procedure and uninitialised, we disallow references to $\vec{r}$ in P, and to $\vec{r}_0$ in Q. Hence the definition of the call can be written as follows.

$$\left|\left[\mathbf{var}\, \vec{r}' : \vec{T}; \mathbf{idle}; \{P\}; \infty \vec{r}', \vec{x}: \left[Q \left[\frac{\vec{r}'}{\vec{r}} \right] \right]; \vec{a}r := \vec{r}'; \mathbf{idle} \right]\right| \tag{15}$$

Because the body of the procedure establishes $Q[\vec{r}\backslash\vec{r}']$ and $\vec{a}r$ is assigned $\vec{r}'$, it also establishes $Q[\vec{r}\backslash\vec{a}r]$ provided $\vec{a}r$ does not occur free in Q.

The following law refines a specification to the local variable block (15), which is equivalent to **call** $pr(\vec{a}r)$. The law does not include the refinement to a call so that it can be reused in the general law in Section 4.4.

Law 35 (*Result Parameters*). *Given an environment, ρ, in which $\vec{r}$ is a vector of distinct variables that do not occur free in P or $\vec{x}$, $\vec{r}_0$ do not occur free in Q, $\vec{a}r$ is a vector of distinct local variables in ρ that are of the same type as $\vec{r}$ (if $\vec{r}$ contains auxiliary parameters then the corresponding actual parameters within $\vec{a}r$ must also be auxiliary), $\vec{a}r$ do not occur free in Q,*[7] *P is an idle-invariant predicate, and Q is a pre- and post-idle-invariant relation in context P, then*

$$\{P\}; \infty \vec{a}r, \vec{x}: \left[Q \left[\vec{r}\backslash\vec{a}r \right] \right] \sqsubseteq_\rho \left|\left[\mathbf{var}\, \vec{r}' : \vec{T}; \mathbf{idle}; \{P\}; \infty\, \vec{r}', \vec{x}: \left[Q \left[\frac{\vec{r}'}{\vec{r}} \right] \right]; \vec{a}r := \vec{r}'; \mathbf{idle} \right]\right|$$

[7] An exception may be made for corresponding elements of $\vec{a}r$ and $\vec{r}$ of the same name.

Proof. The proof relies on the fact that if Q is pre- and post-idle-invariant then so is $Q\left[\vec{r}\backslash\vec{ar}\right]$. That follows from Theorem 33 because local variables are idle-constant expressions.

$$
\begin{array}{ll}
 & \{P\};\ \infty\vec{ar}, \vec{x}{:}\left[Q\left[\vec{r}\backslash\vec{ar}\right]\right] \\
\sqsubseteq_\rho & \text{by Law 29 (local variables idle-invariant)} \\
 & \left|\left[\ \mathbf{var}\,\vec{r}' : \vec{T};\ \{P\};\ \infty\vec{r}', \vec{ar}, \vec{x}{:}\left[Q\left[\vec{r}\backslash\vec{ar}\right]\right]\ \right]\right| \\
\sqsubseteq_\rho & \text{Law 27 (idle-invariant specification)} \\
 & \left|\left[\ \mathbf{var}\,\vec{r}' : \vec{T};\ \mathbf{idle};\ \{P\};\ \infty\vec{r}', \vec{ar}, \vec{x}{:}\left[Q\left[\vec{r}\backslash\vec{ar}\right]\right];\ \mathbf{idle}\ \right]\right| \\
\sqsubseteq_\rho & \text{Law 36 (following assignment) below; property (3) and } \vec{ar} \text{ not free in } Q \\
 & \left|\left[\ \mathbf{var}\,\vec{r}' : \vec{T};\ \mathbf{idle};\ \{P\};\ \infty\,\vec{r}', \vec{x}{:}\left[Q\left[\frac{\vec{r}'}{\vec{r}}\right]\right];\ \vec{ar} := \vec{r}';\ \mathbf{idle}\ \right]\right|
\end{array}
$$

The following law is an adaptation of Law 3.5 from Morgan [29]. Its proof is similar to that of Law 26 (post-idle-invariant postcondition).

Law 36 (*Following Assignment*). *Given an environment ρ, a vector of outputs $\vec{o}$, vectors of local variables, $\vec{v}$ and $\vec{x}$; a post-idle-invariant relation, Q; a vector of idle-constant expressions $\vec{E}$, that is assignment compatible with $\vec{v}$; the following holds*

$$\infty\vec{v}, \vec{x}, \vec{o}{:}\,[Q] \sqsubseteq \infty\vec{v}, \vec{x}, \vec{o}{:}\left[Q\left[\vec{v}\backslash\vec{E}\ @\ \tau\right]\right];\ \vec{v} := \vec{E}.$$

In Law 35 (result parameters) we require that $\vec{ar}$ does not occur in Q to avoid aliasing problems that lead to invalid refinements. For example, assume procedure *par* is defined as

$$\mathbf{procedure}\ par(\mathbf{result}\, r : int)\ \ r{:}\,[r = ar + 1]$$

Without the restriction one could get the following invalid refinement of an infeasible specification (with a false postcondition) to a feasible procedure call that increments ar.

$$
\begin{array}{ll}
 & ar{:}\,[ar = ar + 1] \\
\not\sqsubseteq_\rho & |[\ \mathbf{var}\, r' : T;\ \mathbf{idle};\ \infty r'{:}\,[r' = ar + 1];\ ar := r;\ \mathbf{idle}\ ]| \\
\sqsubseteq_\rho & \mathbf{call}\ \ par(ar)
\end{array}
$$

4.4. General procedure call law

We can combine the above laws for value and result parameters into a general law that also considers input and output parameters. Consider a procedure p with a vector of distinct formal parameters $\vec{d}$, and let $\vec{i}$, $\vec{o}$, $\vec{v}$ and $\vec{r}$ be the sub-vectors of input, output, value and result parameter names, and let the vector of global variables $\vec{x}$ be disjoint from $\vec{d}$.

$$\mathbf{procedure}\ p(\vec{d})\ \ \{P\};\ \infty\vec{v}, \vec{r}, \vec{o}, \vec{x}{:}\,[Q] \tag{16}$$

Given suitable actual parameters $\vec{ai}$, $\vec{ao}$, $\vec{av}$, and $\vec{ar}$, that satisfy the constraints (repeated in the law below) in Definition 16 (procedure call) a call on p satisfies

$$
\begin{array}{ll}
 & \mathbf{call}\ p\left(\hat{d}\left[\dfrac{\vec{ai}, \vec{ao}, \vec{av}, \vec{ar}}{\vec{i}, \vec{o}, \vec{v}, \vec{r}}\right]\right) \\
\sqsupseteq_\rho & |[\ \mathbf{var}\,\vec{v}' : \vec{T}v;\ \mathbf{var}\,\vec{r}' : \vec{T}r;\ \mathbf{idle};\ \vec{v}' := \vec{av};\ \{P'\};\ \infty\vec{v}', \vec{r}', \vec{ao}, \vec{x}{:}\,[Q'];\ \vec{ar} := \vec{r}';\ \mathbf{idle}\ ]|
\end{array}
$$

where $P' \mathrel{\widehat{=}} P\left[\frac{\vec{ai},\vec{ao},\vec{v}',\vec{r}'}{\vec{i},\vec{o},\vec{v},\vec{r}}\right]$ and $Q' \mathrel{\widehat{=}} Q\left[\frac{\vec{ai},\vec{ao},\vec{v}'_0,\vec{v}',\vec{r}'_0,\vec{r}'}{\vec{i},\vec{o},\vec{v}_0,\vec{v},\vec{r}_0,\vec{r}}\right]$.

The following law allows one to refine a specification to a call on p. Its proof makes use of the laws for value and result parameters given in the previous two sections.

Law 37 (*Procedure Call*). *Given an environment, ρ, in which p is defined as in (16), P is an idle-invariant predicate, Q is a pre- and post-idle-invariant relation in context P, $\vec{r}$ does not occur free in P, $\vec{r}_0$ does not occur free in Q, $\vec{ai}$ is a vector of inputs of the same type as $\vec{i}$, $\vec{ao}$ is a vector of distinct outputs of the same type as $\vec{o}$, that do not occur free*

in $\vec{x}$, P and Q,[8] *$\vec{av}$ is a vector of idle-constant expressions of the same type as $\vec{v}$, (non-auxiliary parameters must not contain references to τ, auxiliaries or inputs), $\vec{ar}$ is a vector of distinct variables of the same type as $\vec{r}$ (for auxiliary parameters the actual parameter must also be auxiliary), and $\vec{ar}$ does not occur free in Q,*[9] *then*

$$\left\{ P\left[\frac{\vec{ai}, \vec{ao}, \vec{av} @ \tau}{\vec{i}, \vec{o}, \vec{v}}\right] \wedge def(\vec{av} @ \tau) \right\}; \infty \vec{ar}, \vec{ao}, \vec{x}: \left[\left(\exists \vec{v} : \vec{T}v \bullet Q\left[\frac{\vec{ai}, \vec{ao}, \vec{av}_0 @ \tau_0, \vec{ar}}{\vec{i}, \vec{o}, \vec{v}_0, \vec{r}}\right]\right)\right]$$
$$\sqsubseteq_\rho \textbf{call}\, p\left(\hat{d}\left[\frac{\vec{ai}, \vec{ao}, \vec{av}, \vec{ar}}{\vec{i}, \vec{o}, \vec{v}, \vec{r}}\right]\right).$$

Proof. We define the following abbreviations, in which P' and Q' are defined as above, and $\vec{v}''$ and $\vec{r}''$ are vectors of fresh variable names.

$$P'' \mathrel{\widehat{=}} P'\left[\frac{\vec{v}''}{\vec{v}'}\right]$$
$$Q'' \mathrel{\widehat{=}} Q'\left[\frac{\vec{ar}}{\vec{r}'}\right]\left[\frac{\vec{v}''_0, \vec{v}''}{\vec{v}'_0, \vec{v}'}\right]$$

Given the restrictions in the law, we note that P' and P'' are idle invariant, and Q' and Q'' are pre- and post-idle-invariant. First we show that

$$P''\left[\frac{\vec{av} @ \tau}{\vec{v}''}\right]$$
$$\equiv P\left[\frac{\vec{ai}, \vec{ao}, \vec{v}', \vec{r}'}{\vec{i}, \vec{o}, \vec{v}, \vec{r}}\right]\left[\frac{\vec{v}''}{\vec{v}'}\right]\left[\frac{\vec{av} @ \tau}{\vec{v}''}\right]$$
$\equiv$ by (2) as $\vec{v}''$ not free in P, $\vec{ai}$, $\vec{ao}$, $\vec{v}'$ and $\vec{r}'$
$$P\left[\frac{\vec{ai}, \vec{ao}, \vec{v}', \vec{r}'}{\vec{i}, \vec{o}, \vec{v}, \vec{r}}\right]\left[\frac{\vec{av} @ \tau}{\vec{v}'}\right]$$
$\equiv$ by (2) as $\vec{v}'$ not free in P, $\vec{ai}$, $\vec{ao}$ and $\vec{r}'$; by (1) as $\vec{r}$ not free in P
$$P\left[\frac{\vec{ai}, \vec{ao}, \vec{av} @ \tau}{\vec{i}, \vec{o}, \vec{v}}\right]$$

and then that

$$\left(\exists \vec{v}'' : \vec{T}v \bullet Q''\left[\frac{\vec{av}_0 @ \tau_0}{\vec{v}''_0}\right]\right)$$
$$\equiv \left(\exists \vec{v}'' : \vec{T}v \bullet Q\left[\frac{\vec{ai}, \vec{ao}, \vec{v}'_0, \vec{v}', \vec{r}'_0, \vec{r}'}{\vec{i}, \vec{o}, \vec{v}_0, \vec{v}, \vec{r}_0, \vec{r}}\right]\left[\frac{\vec{ar}}{\vec{r}'}\right]\left[\frac{\vec{v}''_0, \vec{v}''}{\vec{v}'_0, \vec{v}'}\right]\left[\frac{\vec{av}_0 @ \tau_0}{\vec{v}''_0}\right]\right)$$
$\equiv$ by (1) as $\vec{r}_0$ not free in Q; by (2) as $\vec{r}'$ not free in Q, $\vec{ai}$, $\vec{ao}$, $\vec{v}'_0$ and $\vec{v}'$
$$\left(\exists \vec{v}'' : \vec{T}v \bullet Q\left[\frac{\vec{ai}, \vec{ao}, \vec{v}'_0, \vec{v}', \vec{ar}}{\vec{i}, \vec{o}, \vec{v}_0, \vec{v}, \vec{r}}\right]\left[\frac{\vec{v}''_0, \vec{v}''}{\vec{v}'_0, \vec{v}'}\right]\left[\frac{\vec{av}_0 @ \tau_0}{\vec{v}''_0}\right]\right)$$
$\equiv$ by (2) as $\vec{v}'_0$ and $\vec{v}'$ not free in Q, $\vec{ai}$, $\vec{ao}$, $\vec{ar}$
$$\left(\exists \vec{v}'' : \vec{T}v \bullet Q\left[\frac{\vec{ai}, \vec{ao}, \vec{v}''_0, \vec{v}'', \vec{ar}}{\vec{i}, \vec{o}, \vec{v}_0, \vec{v}, \vec{r}}\right]\left[\frac{\vec{av}_0 @ \tau_0}{\vec{v}''_0}\right]\right)$$
$\equiv$ by (2) as $\vec{v}''_0$ not free in Q, $\vec{ai}$, $\vec{ao}$, $\vec{v}''$, $\vec{ar}$
$$\left(\exists \vec{v}'' : \vec{T}v \bullet Q\left[\frac{\vec{ai}, \vec{ao}, \vec{av}_0 @ \tau_0, \vec{v}'', \vec{ar}}{\vec{i}, \vec{o}, \vec{v}_0, \vec{v}, \vec{r}}\right]\right)$$

[8] An exception can be made for names in $\vec{ao}$ that are the same as the corresponding names in $\vec{o}$.
[9] An exception can be made for names in $\vec{ar}$ that are the same as the corresponding names in $\vec{r}$.

$\equiv$ renaming bound variable as $\vec{v}''$ not free in $Q, \vec{ai}, \vec{ao}, \vec{av}_0 @ \tau_0, \vec{ar}$

$$\left(\exists \vec{v} : \vec{T}v \bullet Q\left[\frac{\vec{ai}, \vec{ao}, \vec{av}_0 @ \tau_0, \vec{ar}}{\vec{i}, \vec{o}, \vec{v}_0, \vec{r}}\right]\right)$$

Hence the specification in the law is equivalent to the specification in the first step below.

$$\left\{P''\left[\frac{\vec{av} @ \tau}{\vec{v}''}\right] \wedge def(\vec{av} @ \tau)\right\}; \infty \vec{ar}, \vec{ao}, \vec{x}: \left[\exists \vec{v}'' : \vec{T}v \bullet Q''\left[\frac{\vec{av}_0 @ \tau_0}{\vec{v}''_0}\right]\right]$$

$\sqsubseteq_\rho$ by Law 34 (value parameters)

$$\left|\left[\ \mathbf{var}\ \vec{v}' : \vec{T}v; \mathbf{idle}; \vec{v}' := \vec{av}; \left\{P''\left[\frac{\vec{v}'}{\vec{v}''}\right]\right\}; \infty \vec{v}', \vec{ar}, \vec{ao}, \vec{x}: \quad \left[Q''\left[\frac{\vec{v}'_0, \vec{v}'}{\vec{v}''_0, \vec{v}''}\right]\right]; \mathbf{idle}\ \right]\right|$$

$\sqsubseteq_\rho$ by (2) as $\vec{v}''$ not free in P'; $\vec{v}''_0, \vec{v}''$ not free in $Q'\left[\frac{\vec{ar}}{\vec{r}'}\right]$; $\vec{r}''$ not free in Q'

$$\left|\left[\ \mathbf{var}\ \vec{v}' : \vec{T}v; \mathbf{idle}; \vec{v}' := \vec{av}; \{P'\}; \infty \vec{v}', \vec{ar}, \vec{ao}, \vec{x}: \quad \left[Q'\left[\frac{\vec{r}''}{\vec{r}'}\right]\left[\frac{\vec{ar}}{\vec{r}''}\right]\right]; \mathbf{idle}\ \right]\right|$$

$\sqsubseteq_\rho$ by Law 35 (result parameters) on inner spec. with $Q'\left[\frac{\vec{r}''}{\vec{r}'}\right]$ for Q

$$\left|\left[\ \mathbf{var}\ \vec{v}' : \vec{T}v; \mathbf{idle}; \vec{v}' := \vec{av}; \left|\left[\ \mathbf{var}\ \vec{r}' : \vec{T}r; \mathbf{idle}; \{P'\}; \right.\right.\right.$$

$$\left.\left.\left. \infty \vec{r}', \vec{v}', \vec{ar}, \vec{ao}, \vec{x}: \quad \left[Q'\left[\frac{\vec{r}''}{\vec{r}'}\right]\left[\frac{\vec{r}'}{\vec{r}''}\right]\right]; \vec{ar} := \vec{r}'; \mathbf{idle}\ \right]\right|; \mathbf{idle}\ \right]\right|$$

$\sqsubseteq_\rho$ Law 12(c); $\vec{r}'$ not free in $\vec{v}'$ and $\vec{av}$; $\vec{r}''$ not free in Q'

$$|[\ \mathbf{var}\ \vec{v}' : \vec{T}v; \mathbf{var}\ \vec{r}' : \vec{T}r; \mathbf{idle}; \vec{v}' := \vec{av}; \mathbf{idle}; \{P'\}; \infty \vec{r}', \vec{v}', \vec{ar}, \vec{ao}, \vec{x}: [Q']; \vec{ar} := \vec{r}'; \mathbf{idle}; \mathbf{idle}\ ]|$$

$\sqsubseteq_\rho$ as **idle** $\sqsubseteq_\rho$ **skip** and **idle**; **idle** $\sqsubseteq_\rho$ **idle**

$$|[\ \mathbf{var}\ \vec{v}' : \vec{T}v; \mathbf{var}\ \vec{r}' : \vec{T}r; \mathbf{idle}; \vec{v}' := \vec{av}; \{P'\}; \infty \vec{r}', \vec{v}', \vec{ar}, \vec{ao}, \vec{x}: [Q']; \vec{ar} := \vec{r}'; \mathbf{idle}\ ]|$$

$\sqsubseteq_\rho$ from Definition 16 (procedure call)

$$\mathbf{call}\ p\left(\hat{d}\left[\frac{\vec{ai}, \vec{ao}, \vec{av}, \vec{ar}}{\vec{i}, \vec{o}, \vec{v}, \vec{r}}\right]\right)$$

A variation on Law 37 (procedure call) is if the procedure body does not modify the formal value parameter, $\vec{v}$. In this case $\vec{v}$ can be omitted from the frame in the procedure definition and the existential quantifier is no longer required in the law provided $\vec{v}$ does not occur free in Q. This variant of the law may be used to refine specification (8) given in Section 3 to a procedure call: **call** *Await*(*beam*, *true*, *up_t*, *et*). In this case the formal value parameters *val* and *ev* do not occur free in the postcondition of *Await* (6). The conditions required by the law are:

- the pre-condition of *Await* is idle invariant, which is shown to hold after Definition 18 (idle-invariant);
- the post-condition of *Await* is both pre- and post-idle-invariant, which are shown to hold after Theorem 22 and after Definition 24 (post-idle-invariant), respectively;
- the post-state result parameter *pt* does not occur free in the precondition (5) of *Await*, which holds trivially;
- pt_0 does not occur free in the postcondition of *Await*, which holds trivially;
- the actual value parameter *true* is idle-constant, which holds trivially; and
- the actual (auxiliary) value parameter *up_t* does not occur free in the postcondition of *Await*, which holds trivially.

5. Timing constraint analysis

In order to allow the machine-independent expression of real-time programs we have made use of deadline commands. However, deadline commands cannot be directly implemented. Hence we need to guarantee that every path leading to a deadline command will always reach it before its deadline expires. This process can itself be split into two phases:

- *timing constraint analysis*, that for each non-dead path [13] leading to a deadline, determines a timing constraint that guarantees that the deadline will be met [7]; and
- *worst-case execution-time analysis*, that checks that the worst-case execution time of the code on each path meets its timing constraint [4,31,27].

procedure *Await*(**input** *sensor* : Boolean; **value** *val* : Boolean; **value aux** *ev* : *Time*; **result** *pt* : *time*)
|[**var** p : Boolean; **aux** *before* : *Time*;
repeat
A :: $before := \tau$;
read(*sensor*, p);
B :: **deadline** $ev + 10$ ms;
$\{(p = val \Rightarrow ev \leq \tau) \wedge (p \neq val \Rightarrow before \leq ev) \wedge \tau \leq ev + 10 \text{ ms}\}$
until $p = val$;
$\{ev \leq \tau\}$;
gettime(*pt*);
C :: **deadline** $ev + 10$ ms]|

Fig. 5. Implementation of procedure Await.

The timing constraint analysis phase is machine independent, while the worst-case execution-time analysis depends on the target machine.

The theory of timing constraint analysis is covered in more detail elsewhere [17,25,26,24]. Here we illustrate its interaction with procedures by way of an example. An implementation of procedure *Await* is given in Fig. 5. The labels *A*, *B* and *C* are provided to allow reference to points in the program. It repeatedly tests the value of *sensor* until it changes to equal *val*. From the assumed properties (5) of the *sensor*, when a value equal to *val* is read from *sensor*, the time must be after *ev*. The read must be completed before $ev + 10$ ms in order to ensure that the procedure is not detecting some later change of *sensor* to *val*. Hence the deadline command after the read. If the value read is equal to *val* the loop terminates and one can deduce that *ev* is before the current time, τ. The deadline after the *gettime* ensures that the value of *pt* is a close enough approximation to *ev*. The primitive procedure *gettime* has the following specification.

procedure *gettime*(**result** t : *time*) $t : [\tau_0 \leq t \leq \tau]$

If the repetition never terminates then for any time, t, there is a later time, t', at which both the condition for repeating ($p \neq val$) and the loop invariant (the assertion just before the **until** in Fig. 5) hold, and hence, $t' \leq ev + 10$ ms, holds for arbitrarily large values of t'. Therefore *ev* must be infinity. Note that if the repetition never terminates then the deadline after the loop is never reached and does not have to be considered. The refinement of the specification of procedure *Await* given at the start of Section 3 to the code given in Fig. 5 can be shown using laws for reasoning about (possibly) non-terminating loops given in earlier work [11].

In order for compiled machine code to implement a machine-independent program it must guarantee to meet all its deadlines. The auxiliary variables and parameters aid this analysis. There are two deadlines within procedure *Await* (Fig. 5). The deadline (*B*) within the repetition is reached initially from the entry to the procedure, and subsequently on each iteration. The final deadline (*C*) is reached on exit from the loop. We defer analysis of the entry path to the analysis of the calling program because the context of the calling program is necessary for this, and consider the path (shown in Fig. 6) which must meet the deadline at *C*. It starts at the assignment to *before* (*A*), reads the value of *sensor* into p, passes through the deadline (*B*), loops back to the start of the **repeat** because p is not equal to *val*, performs the assignment to *before* (*A*), reads the value of *sensor*, passes through the deadline (*B*), exits the loop because p is equal to *val*, and assigns the current time to *pt*, before reaching the final deadline (*C*). Evaluation of the guard is either represented by $\varnothing$: $[p = val]$ to indicate exit from the loop, or $\varnothing$: $[p \neq val]$ to indicate that the path followed is to repeat the loop if $p \neq val$.

This path contains two iterations of the loop body. The first iteration reads a *sensor* value not equal to *val* and hence before the time *ev* (i.e., $before \leq ev$), and the second iteration reads a value equal to *val* and hence after *ev*. The initial time assigned to *before*, i.e., the time at which the path begins execution, must be before time *ev* because the value of

A :: $before := \tau$;
  $read(sensor, p)$;
B :: **deadline** $ev + 10$ ms;
  $\{(p = val \Rightarrow ev \leq \tau) \wedge (p \neq val \Rightarrow before \leq ev) \wedge \tau \leq ev + 10 \text{ ms}\}$
  $\varnothing$: $[p \neq val]$;
  -- **repeat**
A :: $before := \tau$;
  $read(sensor, p)$;
B :: **deadline** $ev + 10$ ms;
  $\{(p = val \Rightarrow ev \leq \tau) \wedge (p \neq val \Rightarrow before \leq ev) \wedge \tau \leq ev + 10 \text{ ms}\}$
  $\varnothing$: $[p = val]$;
  $\{ev \leq \tau\}$;
  $gettime(pt)$;
C :: **deadline** $ev + 10$ ms

Fig. 6. Repetition exit path in Await.

$$\left\{\begin{array}{l} down\_t + 10 \text{ ms} \leq up\_t \wedge \\ (beam = true) \textbf{ over } (\tau \ldots down\_t) \cup (up\_t \ldots up\_t + 10 \text{ ms}) \wedge \\ (beam = false) \textbf{ over } (down\_t \ldots up\_t) \end{array}\right\}$$

D :: $\{\tau \leq down\_t\}$;
  |[**var** st, et : *natural* ms; $size$: *natural* mm;
    E :: **call** $Await(beam, false, down\_t, st)$; -- start of object at $down\_t$
    F :: **call** $Await(beam, true, up\_t, et)$; -- end of object at up_t
      $size := (et - st) * velocity$;
      $largeBin := (limit \leq size)$;
    G :: **deadline** $up\_t + 20$ ms]|

Fig. 7. Calling program.

p was not equal to val on the first evaluation of the guard. The final deadline on the path is $ev + 10$ ms. Hence, if the path is guaranteed to execute in less than 10 ms, it will always meet its deadline.

For deadline B, other than on the initial entry into the procedure, we need to consider a path that is the same as that in Fig. 6 up until the second occurrence of deadline B. This subpath also has a timing constraint of 10 ms and hence is subsumed by the longer path. Analysing this (shorter) path effectively covers any possible number of iterations of the loop (see [17] for more details). Assuming the deadline is met on first entry, this path gives the timing constraint to show that if the deadline is met on one iteration, it will be met on the next iteration.

The calling program. The analysis of the calling program has to take into account deadlines within the procedure calls. Consider the program in Fig. 7 that makes two calls on the procedure *Await*. Its task is to determine the size of an object that passes through a light beam and to control a Boolean actuator, *largeBin*, that determines whether the object goes in the bin for large objects or that for small objects. The variable *beam* is *true* unless there is an object blocking the beam of light, in which case it is *false*. We use the auxiliaries $down\_t$ and up_t to stand for the times

D :: $\{\tau \leq down\_t\}$;
alloc var st, et : *natural* ms; $size$: *natural* mm;
E :: -- **call** $Await(beam, false, down\_t, st)$;
alloc var val : Boolean;
alloc aux ev : *Time*;
alloc var pt : *time*;
$val, ev := false, down\_t$;
alloc var p : Boolean;
alloc aux $before$: *Time*;
-- **repeat**
$E.A$ :: $before := \tau$;
$read(beam, p)$;
$E.B$ :: **deadline** $ev + 10$ ms;

Fig. 8. Calling path.

the value of *beam* changes from true to false and from false to true, respectively. The local variables st and et capture the approximate start and end times when an object passes the light beam, and variable $size$ is used to calculate the (approximate) size of the object from the time it took to pass. (As part of the declaration of the types of variables we also declare their units [14].) If the calculated size is greater than or equal to *limit* then *largeBin* is set to *true*, otherwise it is set to *false*. It is assumed that the program starts before the object reaches the beam (i.e., $\tau \leq down\_t$).

There is a path (shown in Fig. 8) that starts at D in Fig. 7. The path allocates some local variables and then makes the first call (E) to *Await*. We use the notation "**alloc var** $v : T$", to abbreviate "**alloc var** v; v: $[v \in T]$", and similarly for auxiliaries. The call allocates the local and auxiliary variables corresponding to the formal value and result parameters, assigns the formal value parameters the actual value expressions, allocates the local variable p, extends the auxiliaries with *before*, and follows the path into the repetition, ending at the first deadline (B) of $ev + 10$ ms. The deadline is labelled $E.B$ to indicate that it is the deadline labelled B within the call to *Await* labelled E. The initial assertion guarantees the start time of the path is less than or equal to $down\_t$. For this call to *Await*, ev is $down\_t$ and hence the final deadline ($E.B$) is $down\_t + 10$ ms. Therefore a suitable constraint on the path is 10 ms.

There is another path starting at the final deadline (C) within *Await* on its first call (E). For the first call ev is $down\_t$ and hence the deadline ($E.C$) is $down\_t + 10$ ms. The path exits the first call, assigning the result variable and deallocating local variables and parameters as part of this, and then enters the second call (F) following a path similar to that in Fig. 8 and ending at the first deadline ($F.B$). For the second call ev is up_t and hence the deadline ($F.B$) is $up\_t + 10$ ms. The overall constraint on the path is

$$up\_t + 10 \text{ ms} - (down\_t + 10 \text{ ms}) = up\_t - down\_t$$

which was assumed to be no less than 10 ms (in the assumption in Fig. 7).

The other path that crosses a procedure boundary starts from the final deadline in the second call ($F.C$), which has a deadline of $up\_t + 10$ ms. It then exits the procedure (assigning pt to et and deallocating the local state and formal parameters), calculates $size$ and assigns to *largeBin* before reaching the final deadline (G) in the calling program, which has a deadline of $up\_t + 20$ ms. The constraint on this path is 10 ms. Note that if the final deadline (G) was $up\_t + 10$ ms, the constraint would be 0 ms and hence infeasible. In this case we would need to consider a longer path that starts earlier. It would be similar to the path in Fig. 6 followed by the path just described. The constraint on that whole path would be 10 ms.

Once timing constraints on paths have been determined the final step is to verify that the machine code corresponding to each path is always executed within the timing constraint on the target machine. Such worst-case execution-time analysis of machine code is treated elsewhere in the literature [4,27,31].

6. Conclusions

Our aim was to develop laws for refining real-time programs using procedures, with the laws as similar as possible to those for the non-real-time case [1,29,6,34]. Procedure entry and exit mechanisms incur a time overhead, which needs to be accounted for in the refinement laws. We achieve that by insisting that the preconditions and postconditions of specifications of procedures satisfy idle-invariance properties. In many cases showing that predicates are idle invariant is trivial because they do not refer to the current time variable, τ. The remaining cases that do refer to τ are the interesting ones that need to be checked.

We allow value and result parameters similar to those of Morgan [28,29]. As usual, restrictions on parameters are required to avoid aliasing problems [21]. In order to allow timing constraints to be expressed, we allow auxiliary procedure parameters to be used. Auxiliaries are used only to allow reasoning about the behaviour (especially timing behaviour) of the program and no machine code is generated for any uses of auxiliaries. External input and output parameters differ from value and result parameters in that the values of input and output parameters are observable over the duration of the procedure call, not just on entry and exit. The implementation of such parameters is expected to be by reference.

Although we have introduced four (or six if one differentiates auxiliary parameters) different forms of parameters, each of these can be justified. The differentiation between external input/outputs and other variables is necessary because their intermediate values are observable. Auxiliaries are justified on the basis that they allow timing constraints that cross procedure boundaries to be expressed in a simple manner.

Our overall goal is to allow the derivation of machine-independent programs from specifications of the required behaviour. This is facilitated by the use of deadline commands, auxiliary variables, and for procedures, auxiliary parameters. These allow timing constraints embedded in the original specification to be explicitly stated within the program. The implementation of a machine-independent program on a target machine requires that one determines suitable timing constraints on paths leading to deadlines that ensure that the deadline will always be met – this process is also machine independent – and then analyse the compiled machine code for each path to ensure its possible execution times on the target machine meet its timing constraint. The partitioning of the development of a program into a machine-independent part followed by a machine-dependent worst-case execution-time analysis, has the advantage that only the latter part needs to be redone to retarget the program to a different machine (or a model with different performance). This has long term benefits in maintaining critical real-time software over multiple platforms and new generations of hardware.

Acknowledgements

This research was supported by Australian Research Council (ARC) Discovery Grant DP0558408, *Analysing and generating fault-tolerant real-time systems*. I would like to thank David Carrington, Colin Fidge, and Luke Wildman for feedback on earlier drafts of this paper.

References

[1] R.-J. Back, J. von Wright, Refinement Calculus: A Systematic Introduction, Springer, 1998.
[2] J. Barnes, High Integrity Ada: The Spark Approach, Addison Wesley, 1997.
[3] A. Burns, B. Dobbing, G. Romanski, The Ravenscar tasking profile for high integrity real-time programs, in: L. Asplund (Ed.), Reliable Software Technologies—Ada-Europe '98, in: LNCS, vol. 1411, Springer-Verlag, 1998, pp. 263–275.
[4] R. Chapman, A. Burns, A.J. Wellings, Integrated program proof and worst-case timing analysis of SPARK Ada, in: ACM Workshop on Language, Compiler and Tool Support for Real-time Systems, ACM Press, 1994.
[5] C.J. Fidge, I.J. Hayes, G. Watson, The deadline command, IEE Proceedings—Software 146 (2) (1999) 104–111.
[6] L. Groves, Procedures in the refinement calculus: A new approach? in: J. He, J. Cooke, P. Wallis (Eds.), BCS-FACS Seventh Refinement Workshop, Electronic Workshops in Computing, Springer-Verlag, 1996.
[7] S. Grundon, I.J. Hayes, C.J. Fidge, Timing constraint analysis, in: C. McDonald (Ed.), Computer Science '98: Proc. 21st Australasian Computer Sci. Conf., ACSC'98, Springer, 1998, pp. 575–586.
[8] I.J. Hayes (Ed.), Specification Case Studies, Prentice Hall, 1987. 2nd edition 1993.
[9] I.J. Hayes, Real-time program refinement using auxiliary variables, in: M. Joseph (Ed.), Proc. Formal Techniques in Real-Time and Fault-Tolerant Systems, in: LNCS, vol. 1926, Springer, 2000, pp. 170–184.
[10] I.J. Hayes, Reasoning about real-time programs using idle-invariant assertions, in: J.S. Dong, J. He, M. Purvis (Eds.), Proceedings 7th Asia-Pacific Software Engineering Conference, APSEC 2000, IEEE Computer Society, 2000, pp. 16–23.

[11] I.J. Hayes, Reasoning about real-time repetitions: Terminating and nonterminating, Science of Computer Programming 43 (2–3) (2002) 161–192.
[12] I.J. Hayes, A predicative semantics for real-time refinement, in: A. McIver, C.C. Morgan (Eds.), Programming Methodology, Springer Verlag, 2003, pp. 109–133.
[13] I.J. Hayes, C.J. Fidge, K. Lermer, Semantic characterisation of dead control-flow paths, IEE Proceedings—Software 148 (6) (2001) 175–186. December.
[14] I.J. Hayes, B.P. Mahony, Using units of measurement in formal specifications, Formal Aspects of Computing 7 (3) (1995) 329–347.
[15] I.J. Hayes, M. Utting, Coercing real-time refinement: A transmitter, in: D.J. Duke, A.S. Evans (Eds.), BCS-FACS Northern Formal Methods Workshop, NFMW'96, in: Electronic Workshops in Computing, Springer, 1997.
[16] I.J. Hayes, M. Utting, A sequential real-time refinement calculus, Acta Informatica 37 (6) (2001) 385–448.
[17] I.J. Hayes, Programs as paths: An approach to timing constraint analysis, in: J.S. Dong, J. Woodcock (Eds.), Formal Methods and Software Engineering: Proc. 5th Int. Conf. on Formal Engineering Methods, ICFEM 2003, in: Lecture Notes in Computer Science, vol. 2885, Springer Verlag, 2003, pp. 1–15.
[18] E.C.R. Hehner, Termination is timing, in: J.L.A. van de Snepscheut (Ed.), Mathematics of Program Construction, in: Lecture Notes in Computer Science, vol. 375, Springer, June 1989, pp. 36–47.
[19] E.C.R. Hehner, A Practical Theory of Programming, Springer, 1993.
[20] W.H. Hesselink, Predicate transformers for recursive procedures with local variables, Formal Aspects of Computing 11 (6) (1999) 616–636.
[21] C.A.R. Hoare, Procedures and parameters: An axiomatic approach, in: E. Engeler (Ed.), Symposium on Semantics of Algorithmic Languages, in: Lecture Notes in Mathematics, vol. 188, Springer-Verlag, 1971, pp. 102–116. Reprinted in [22, Chapter 6].
[22] C.A.R. Hoare, in: C.B. Jones (Ed.), Essays in Computing Science, Prentice Hall International, 1989.
[23] J. Hooman, O. van Roosmalen, An approach to platform independent real-time programming: (1) formal description, Real-Time Systems 19 (1) (2000) 61–85.
[24] K. Lermer, C.J. Fidge, I.J. Hayes, Formal semantics for program paths, in: J. Harland (Ed.), Computing: The Australian Theory Symposium (CATS) 2003, in: Electronic Notes in Theoretical Computer Science (ENTCS), vol. 78, Elsevier, February 2003, pp. 1–24.
[25] K. Lermer, C.J. Fidge, I.J. Hayes, A theory for execution time derivation in real-time programs, Theoretical Computer Science 346 (1) (2005) 3–27.
[26] K. Lermer, C.J. Fidge, I.J. Hayes, Linear approximation of execution-time constraints, Formal Aspects of Computing 15 (4) (2003) 319–348.
[27] S.-S. Lim, Y.H. Bae, G.T. Jang, B.-D. Rhee, S.L. Min, C.Y. Park, H. Shin, K. Park, S.-M. Moon, C.S. Kim, An accurate worst case timing analysis for RISC processors, IEEE Transactions on Software Engineering 21 (7) (1995) 593–604.
[28] C.C. Morgan, Procedures, parameters, and abstraction: Separate concerns, Science of Computer Programming 11 (1) (1988) 17–28. Reprinted in [30, pp. 47–58].
[29] C.C. Morgan, Programming from Specifications, 2nd edition, Prentice Hall, 1994.
[30] C.C. Morgan, T.N. Vickers (Eds.), On the Refinement Calculus, Springer-Verlag, 1994.
[31] C.Y. Park, Predicting program execution times by analyzing static and dynamic program paths, Real-Time Systems 5 (1993) 31–62.
[32] A.C. Shaw, Reasoning about time in higher-level language software, IEEE Transactions on Software Engineering 15 (7) (1989) 875–889.
[33] J.M. Spivey, The Z Notation: A Reference Manual, 2nd edition, Prentice Hall International, 1992.
[34] M. Staples, Interfaces for refining recursion and procedures, Formal Aspects of Computing 12 (5) (2000) 372–391.
[35] L.P. Wildman, C.J. Fidge, D.A. Carrington, The variety of variables in automated real-time refinement, Formal Aspects of Computing 15 (2003) 258–279.